

109

ic518 U.S. PTO

09/275766



03/25/99

# APPENDIX N

COPYRIGHT 1998, LANGUAGE ANALYSIS SYSTEMS, INC.

# NAMEHUNTER

```
//      File: NH_util.cpp
//
//      Description:
//
//          Implementation of various utility functions used in the SNAPi
//
//
//      History:
//
//          5/15/97      EFB      Created
//          3/20/98      EFB      Changed names to NH from SN
//
```

```
#include    <string.h>
```

```
#include    "NH_util.hpp"
```

```
#include    "NHCompParms.hpp"
```

```
//      function to remove leading and trailing spaces from a string
//      in place.
```

```
// Strips the string at either end or both ends.
// Stripchars specify the characters that should
// be stripped. We start by seeing if they want the
// trailing chars stripped, which is easy. We simply
// work backwards from the end of the string, looking for
// the first non-strippable character, and terminate the
// string just past that character. Then if they wanted
// leading chars stripped, we work forwards to the first
// non-strippable char, and then move that and each following
// char to the beginning of the string.
```

```
void NH_strip(char *aString)
```

```
{
```

```
    char *end_point;
```

```
    char *ch;
```

```
    int len;
```

```
    if ((len = strlen(aString)) != 0) { // if there is a string
```

```
        // start at end
```

```
        end_point = aString + len - 1;
```

```

        // ,and work back till we get a non-space or get to
        // the begining of our string, chopping off what's left.
        // Also make sure we don't zoom right past the beginning of the
        // string.
        for (; strchr(NH_DEFAULT_WHITESPACE, *end_point) != NULL &&
end_point != aString; end_point--)
        ;
        // if string was all whitespace
        if ((end_point == aString) && strchr(NH_DEFAULT_WHITESPACE,
*aString) != NULL)
            *aString = EOS; // erase it all, and we're done, could return here
        else
            *(end_point + 1) = EOS; // just chop off excess blanks

        // make sure there is still a string, since it might
        // have been stripped entirely above.
        if (*aString) {
            // now find first non space. we know string has at least one
            // nonwhite space, so we don't have to check for NULL.
            for (ch = aString; strchr(NH_DEFAULT_WHITESPACE, *ch) !=
NULL; ch++)
                ;
            if (ch != aString) { // if there were leading spaces, move the block
back
                char *target = aString;
                while (*ch != EOS) {
                    *target = *ch;
                    target++;
                    ch++;
                }
                // and get the null char also
                *target = *ch;
            } // end if (are there leading spaces?)
        } // end if (and text left?)
    } // end (is there a string at all ?)
}

```

```

char *    NH_strchr(char *stringStart, char *searchPos, char searchChar)
{
    while (1)    {
        if (*searchPos == searchChar)
            break;
        if (searchPos == stringStart) {

```



```
        searchPos = NULL;           // string not found, so return
NULL
        break;
    }
    searchPos--;
}
return searchPos;
}
```

```
//
// File: NH_queens_arrays.hpp
//
// Description:
//
//     Contains global definitions and declarations for the valid
//     combinations of indexes for the best score calculation
//
//
// History:
//
//     6/4/97 EFB      Created
//     3/20/98      EFB      Changed names to NH from SN
//
```

```
typedef      unsigned char byte;
```

```
byte  twoByTwo[] = {1, 0,
                                0, 1};
```

```
byte  twoByThree[] = {    1, 2,
    1, 0,
    2, 1,
    2, 0,
    0, 1,
    0, 2};
```

```
byte  twoByFour[] = {    1, 2,
    1, 3,
    1, 0,
    2, 1,
    2, 3,
    2, 0,
```

3, 1,

3, 2,

3, 0,

0, 1,

0, 2,

0, 3};

byte twoByFive[] = { 1, 2,

1, 3,

1, 4,

1, 0,

2, 1,

2, 3,

2, 4,

2, 0,

3, 1,

3, 2,

3, 4,

3, 0,

4, 1,

4, 2,

4, 3,

4, 0,

0, 1,

0, 2,

0, 3,

0, 4};

byte threeByThree[] = { 1, 2, 0,

1, 0, 2,

2, 1, 0,

2, 0, 1,

0, 1, 2,

0, 2, 1};

byte threeByFour[] = { 1, 2, 3,

1, 2, 0,

1, 3, 2,

1, 3, 0,

1, 0, 2,

1, 0, 3,

2, 1, 3,

2, 1, 0,

2, 3, 1,

2, 3, 0,

2, 0, 1,

2, 0, 3,

3, 1, 2,

3, 1, 0,

3, 2, 1,

3, 2, 0,

3, 0, 1,

3, 0, 2,

0, 1, 2,

0, 1, 3,

0, 2, 1,

0, 2, 3,

0, 3, 1,

0, 3, 2};

byte threeByFive[] = { 1, 2, 3,

1, 2, 4,

1, 2, 0,

1, 3, 2,

1, 3, 4,

1, 3, 0,

1, 4, 2,

1, 4, 3,

1, 4, 0,

1, 0, 2,

1, 0, 3,

1, 0, 4,

2, 1, 3,

2, 1, 4,

2, 1, 0,

2, 3, 1,

2, 3, 4,

2, 3, 0,

2, 4, 1,

2, 4, 3,

2, 4, 0,

2, 0, 1,

2, 0, 3,

2, 0, 4,

3, 1, 2,

3, 1, 4,

3, 1, 0,

3, 2, 1,

3, 2, 4,

3, 2, 0,

3, 4, 1,

3, 4, 2,

3, 4, 0,

3, 0, 1,

3, 0, 2,

3, 0, 4,

4, 1, 2,

4, 1, 3,

4, 1, 0,

4, 2, 1,

4, 2, 3,

4, 2, 0,

4, 3, 1,

4, 3, 2,

4, 3, 0,

4, 0, 1,

4, 0, 2,

4, 0, 3,

0, 1, 2,

0, 1, 3,

0, 1, 4,

0, 2, 1,

0, 2, 3,

0, 2, 4,

0, 3, 1,

0, 3, 2,

0, 3, 4,

0, 4, 1,

0, 4, 2,

0, 4, 3};

byte fourByFour[] = { 1, 2, 3, 0,

1, 2, 0, 3,

1, 3, 0, 2,

1, 3, 2, 0,

1, 0, 2, 3,

1, 0, 3, 2,

2, 1, 3, 0,

2, 1, 0, 3,

2, 3, 1, 0,

2, 3, 0, 1,

2, 0, 1, 3,

2, 0, 3, 1,

3, 1, 2, 0,

3, 1, 0, 2,

3, 2, 1, 0,

3, 2, 0, 1,



3, 0, 1, 2,

3, 0, 2, 1,

0, 1, 2, 3,

0, 1, 3, 2,

0, 2, 1, 3,

0, 2, 3, 1,

0, 3, 1, 2,

0, 3, 2, 1});

byte fourByFive[] = { 1, 2, 3, 4,

1, 2, 3, 0,

1, 2, 4, 3,

1, 2, 4, 0,

1, 2, 0, 3,

1, 2, 0, 4,

1, 3, 2, 4,

1, 3, 2, 0,

1, 3, 4, 2,

1, 3, 4, 0,

1, 3, 0, 2,

1, 3, 0, 4,

1, 4, 2, 3,

1, 4, 2, 0,

1, 4, 3, 2,

1, 4, 3, 0,

1, 4, 0, 2,

1, 4, 0, 3,

1, 0, 2, 3,

1, 0, 2, 4,

1, 0, 3, 2,

1, 0, 3, 4,

1, 0, 4, 2,

1, 0, 4, 3,

2, 1, 3, 4,

2, 1, 3, 0,

2, 1, 4, 3,

2, 1, 4, 0,

2, 1, 0, 3,

2, 1, 0, 4,

2, 3, 1, 4,

2, 3, 1, 0,

2, 3, 4, 1,

2, 3, 4, 0,

2, 3, 0, 1,

2, 3, 0, 4,

2, 4, 1, 3,

2, 4, 1, 0,

2, 4, 3, 1,

2, 4, 3, 0,

2, 4, 0, 1,

2, 4, 0, 3,

2, 0, 1, 3,

2, 0, 1, 4,

2, 0, 3, 1,

2, 0, 3, 4,

2, 0, 4, 1,

2, 0, 4, 3,

3, 2, 1, 4,

3, 2, 1, 0,

3, 2, 4, 1,

3, 2, 4, 0,

3, 2, 0, 1,

3, 2, 0, 4,

3, 1, 2, 4,

3, 1, 2, 0,

3, 1, 4, 2,

3, 1, 4, 0,

3, 1, 0, 2,

3, 1, 0, 4,

3, 4, 2, 1,

3, 4, 2, 0,

3, 4, 1, 2,

3, 4, 1, 0,

3, 4, 0, 2,

3, 4, 0, 1,

3, 0, 2, 1,

3, 0, 2, 4,

3, 0, 1, 2,

3, 0, 1, 4,

3, 0, 4, 2,

3, 0, 4, 1,

4, 2, 3, 1,

4, 2, 3, 0,

4, 2, 1, 3,

4, 2, 1, 0,

4, 2, 0, 3,

4, 2, 0, 1,

4, 3, 2, 1,

4, 3, 2, 0,

4, 3, 1, 2,

4, 3, 1, 0,

4, 3, 0, 2,

4, 3, 0, 1,

4, 1, 2, 3,

4, 1, 2, 0,

4, 1, 3, 2,

4, 1, 3, 0,

4, 1, 0, 2,

4, 1, 0, 3,

4, 0, 2, 3,

4, 0, 2, 1,

4, 0, 3, 2,

4, 0, 3, 1,

4, 0, 1, 2,

4, 0, 1, 3,

0, 2, 3, 4,

0, 2, 3, 1,

0, 2, 4, 3,

0, 2, 4, 1,

0, 2, 1, 3,

0, 2, 1, 4,

0, 3, 2, 4,

0, 3, 2, 1,

0, 3, 4, 2,

0, 3, 4, 1,

0, 3, 1, 2,

0, 3, 1, 4,

0, 4, 2, 3,

0, 4, 2, 1,

0, 4, 3, 2,

0, 4, 3, 1,

0, 4, 1, 2,

0, 4, 1, 3,

0, 1, 2, 3,

0, 1, 2, 4,

0, 1, 3, 2,

0, 1, 3, 4,

0, 1, 4, 2,

0, 1, 4, 3}};

byte fiveByFive[] = { 1, 2, 3, 4, 0,

1, 2, 3, 0, 4,

1, 2, 4, 3, 0,

1, 2, 4, 0, 3,

1, 2, 0, 3, 4,

1, 2, 0, 4, 2,

1, 3, 2, 4, 0,

1, 3, 2, 0, 4,

1, 3, 4, 2, 0,

1, 3, 4, 0, 2,

1, 3, 0, 2, 4,

1, 3, 0, 4, 2,

1, 4, 2, 3, 0,

1, 4, 2, 0, 3,

1, 4, 3, 2, 0,

1, 4, 3, 0, 2,

1, 4, 0, 2, 3,

1, 4, 0, 3, 2,

1, 0, 2, 3, 4,

1, 0, 2, 4, 3,

1, 0, 3, 2, 4,

1, 0, 3, 4, 2,

1, 0, 4, 2, 3,

1, 0, 4, 3, 2,

2, 1, 3, 4, 0,

2, 1, 3, 0, 4,

2, 1, 4, 3, 0,

2, 1, 4, 0, 3,

2, 1, 0, 3, 4,

2, 1, 0, 4, 1,

2, 3, 1, 4, 0,

2, 3, 1, 0, 4,

2, 3, 4, 1, 0,

2, 3, 4, 0, 1,

2, 3, 0, 1, 4,

2, 3, 0, 4, 1,

2, 4, 1, 3, 0,

2, 4, 1, 0, 3,

2, 4, 3, 1, 0,

2, 4, 3, 0, 1,

2, 4, 0, 1, 3,

2, 4, 0, 3, 1,

2, 0, 1, 3, 4,

2, 0, 1, 4, 3,

2, 0, 3, 1, 4,

2, 0, 3, 4, 1,

2, 0, 4, 1, 3,

2, 0, 4, 3, 1,



3, 2, 1, 4, 0,

3, 2, 1, 0, 4,

3, 2, 4, 1, 0,

3, 2, 4, 0, 1,

3, 2, 0, 1, 4,

3, 2, 0, 4, 2,

3, 1, 2, 4, 0,

3, 1, 2, 0, 4,

3, 1, 4, 2, 0,

3, 1, 4, 0, 2,

3, 1, 0, 2, 4,

3, 1, 0, 4, 2,

3, 4, 2, 1, 0,

3, 4, 2, 0, 1,

3, 4, 1, 2, 0,

3, 4, 1, 0, 2,

3, 4, 0, 2, 1,

3, 4, 0, 1, 2,

3, 0, 2, 1, 4,

3, 0, 2, 4, 1,

3, 0, 1, 2, 4,

3, 0, 1, 4, 2,

3, 0, 4, 2, 1,

3, 0, 4, 1, 2,

4, 2, 3, 1, 0,

4, 2, 3, 0, 1,

4, 2, 1, 3, 0,

4, 2, 1, 0, 3,

4, 2, 0, 3, 1,

4, 2, 0, 1, 2,

4, 3, 2, 1, 0,

4, 3, 2, 0, 1,

4, 3, 1, 2, 0,

4, 3, 1, 0, 2,

4, 3, 0, 2, 1,

4, 3, 0, 1, 2,

4, 1, 2, 3, 0,

4, 1, 2, 0, 3,

4, 1, 3, 2, 0,

4, 1, 3, 0, 2,

4, 1, 0, 2, 3,

4, 1, 0, 3, 2,

4, 0, 2, 3, 1,

4, 0, 2, 1, 3,

4, 0, 3, 2, 1,

4, 0, 3, 1, 2,

4, 0, 1, 2, 3,

4, 0, 1, 3, 2,

0, 2, 3, 4, 1,

0, 2, 3, 1, 4,

0, 2, 4, 3, 1,

0, 2, 4, 1, 3,

0, 2, 1, 3, 4,

0, 2, 1, 4, 2,

0, 3, 2, 4, 1,

0, 3, 2, 1, 4,

0, 3, 4, 2, 1,

0, 3, 4, 1, 2,

0, 3, 1, 2, 4,

0, 3, 1, 4, 2,

0, 4, 2, 3, 1,

0, 4, 2, 1, 3,

0, 4, 3, 2, 1,

0, 4, 3, 1, 2,

0, 4, 1, 2, 3,

0, 4, 1, 3, 2,

0, 1, 2, 3, 4,

0, 1, 2, 4, 3,

0, 1, 3, 2, 4,

0, 1, 3, 4, 2,

0, 1, 4, 2, 3,

0, 1, 4, 3, 2};

4, 3, 1, 0,  
4, 3, 0, 2,  
4, 3, 0, 1,  
4, 1, 2, 3,  
4, 1, 2, 0,  
4, 1, 3, 2,  
4, 1, 3, 0,  
4, 1, 0, 2,  
4, 1, 0, 3,  
4, 0, 2, 3,  
4, 0, 2, 1,  
4, 0, 3, 2,  
4, 0, 3, 1,  
4, 0, 1, 2,  
4, 0, 1, 3,  
0, 2, 3, 4,  
0, 2, 3, 1,  
0, 2, 4, 3,  
0, 2, 4, 1,  
0, 2, 1, 3,  
0, 2, 1, 4,  
0, 3, 2, 4,  
0, 3, 2, 1,  
0, 3, 4, 2,  
0, 3, 4, 1,  
0, 3, 1, 2,  
0, 3, 1, 4,  
0, 4, 2, 3,  
0, 4, 2, 1,  
0, 4, 3, 2,  
0, 4, 3, 1,

```
0, 4, 1, 2,  
0, 4, 1, 3,  
0, 1, 2, 3,  
0, 1, 2, 4,  
0, 1, 3, 2,  
0, 1, 3, 4,  
0, 1, 4, 2,  
0, 1, 4, 3};
```

```
byte fiveByFive[] = { 1, 2, 3, 4, 0,  
1, 2, 3, 0, 4,  
1, 2, 4, 3, 0,  
1, 2, 4, 0, 3,  
1, 2, 0, 3, 4,  
1, 2, 0, 4, 2,  
1, 3, 2, 4, 0,  
1, 3, 2, 0, 4,  
1, 3, 4, 2, 0,  
1, 3, 4, 0, 2,  
1, 3, 0, 2, 4,  
1, 3, 0, 4, 2,  
1, 4, 2, 3, 0,  
1, 4, 2, 0, 3,  
1, 4, 3, 2, 0,  
1, 4, 3, 0, 2,  
1, 4, 0, 2, 3,  
1, 4, 0, 3, 2,  
1, 0, 2, 3, 4,  
1, 0, 2, 4, 3,  
1, 0, 3, 2, 4,  
1, 0, 3, 4, 2,
```

1, 0, 4, 2, 3,  
1, 0, 4, 3, 2,  
2, 1, 3, 4, 0,  
2, 1, 3, 0, 4,  
2, 1, 4, 3, 0,  
2, 1, 4, 0, 3,  
2, 1, 0, 3, 4,  
2, 1, 0, 4, 1,  
2, 3, 1, 4, 0,  
2, 3, 1, 0, 4,  
2, 3, 4, 1, 0,  
2, 3, 4, 0, 1,  
2, 3, 0, 1, 4,  
2, 3, 0, 4, 1,  
2, 4, 1, 3, 0,  
2, 4, 1, 0, 3,  
2, 4, 3, 1, 0,  
2, 4, 3, 0, 1,  
2, 4, 0, 1, 3,  
2, 4, 0, 3, 1,  
2, 0, 1, 3, 4,  
2, 0, 1, 4, 3,  
2, 0, 3, 1, 4,  
2, 0, 3, 4, 1,  
2, 0, 4, 1, 3,  
2, 0, 4, 3, 1,  
3, 2, 1, 4, 0,  
3, 2, 1, 0, 4,  
3, 2, 4, 1, 0,  
3, 2, 4, 0, 1,

3, 2, 0, 1, 4,

3, 2, 0, 4, 2,

3, 1, 2, 4, 0,

3, 1, 2, 0, 4,

3, 1, 4, 2, 0,

3, 1, 4, 0, 2,

3, 1, 0, 2, 4,

3, 1, 0, 4, 2,

3, 4, 2, 1, 0,

3, 4, 2, 0, 1,

3, 4, 1, 2, 0,

3, 4, 1, 0, 2,

3, 4, 0, 2, 1,

3, 4, 0, 1, 2,

3, 0, 2, 1, 4,

3, 0, 2, 4, 1,

3, 0, 1, 2, 4,

3, 0, 1, 4, 2,

3, 0, 4, 2, 1,

3, 0, 4, 1, 2,

4, 2, 3, 1, 0,

4, 2, 3, 0, 1,

4, 2, 1, 3, 0,

4, 2, 1, 0, 3,

4, 2, 0, 3, 1,

4, 2, 0, 1, 2,

4, 3, 2, 1, 0,

4, 3, 2, 0, 1,

4, 3, 1, 2, 0,

4, 3, 1, 0, 2,

4, 3, 0, 2, 1,



4, 3, 0, 1, 2,  
4, 1, 2, 3, 0,  
4, 1, 2, 0, 3,  
4, 1, 3, 2, 0,  
4, 1, 3, 0, 2,  
4, 1, 0, 2, 3,  
4, 1, 0, 3, 2,  
4, 0, 2, 3, 1,  
4, 0, 2, 1, 3,  
4, 0, 3, 2, 1,  
4, 0, 3, 1, 2,  
4, 0, 1, 2, 3,  
4, 0, 1, 3, 2,  
0, 2, 3, 4, 1,  
0, 2, 3, 1, 4,  
0, 2, 4, 3, 1,  
0, 2, 4, 1, 3,  
0, 2, 1, 3, 4,  
0, 2, 1, 4, 2,  
0, 3, 2, 4, 1,  
0, 3, 2, 1, 4,  
0, 3, 4, 2, 1,  
0, 3, 4, 1, 2,  
0, 3, 1, 2, 4,  
0, 3, 1, 4, 2,  
0, 4, 2, 3, 1,  
0, 4, 2, 1, 3,  
0, 4, 3, 2, 1,  
0, 4, 3, 1, 2,  
0, 4, 1, 2, 3,

0, 4, 1, 3, 2,

0, 1, 2, 3, 4,

0, 1, 2, 4, 3,

0, 1, 3, 2, 4,

0, 1, 3, 4, 2,

0, 1, 4, 2, 3,

0, 1, 4, 3, 2};

```

//      File:  NH_getErrorText.cpp
//
//      Description:
//
//      Implementation to the NH_getErrorText function.  This
function can
//      be used to return the error text for an associated error
code.
//
//
//      History:
//
//      6/23/97      EFB      Created
//      3/20/98      EFB      Changed names to NH from SN
//

#include      "NH_get_error_text.h"

#include      <string.h>

void  NH_get_error_text(NHReturnCode errorCode, char *textBuffer, int
maxChars)
{
    char  *errorMsgPtr;

    switch      (errorCode) {
        case  NH_SUCCESS:
            errorMsgPtr = "Operation successful";
            break;
        case  NH_MATCH:
            errorMsgPtr = "The comparison matched";
            break;
        case  NH_NO_MATCH:
            errorMsgPtr = "The comparison did not match";
            break;
        case  NH_INVALID_SCORE_THRESH:
            errorMsgPtr = "The threshold must be between 0.0 and
1.0";
            break;
        case  NH_INVALID_GN_INIT_SCORE:
            errorMsgPtr = "The GN initial score must be between
0.0 and 1.0";
            break;
        case  NH_INVALID_NH_INIT_SCORE:
            errorMsgPtr = "The SN initial score must be between
0.0 and 1.0";
            break;
        case  NH_INVALID_GN_INIT_ON_INIT_MATCH_SCORE:
            errorMsgPtr = "The GN initial on initial match score
must be between 0.0 and 1.0";
            break;
        case  NH_INVALID_NH_INIT_ON_INIT_MATCH_SCORE:
            errorMsgPtr = "The SN initial on initial match score
must be between 0.0 and 1.0";
            break;
        case  NH_INVALID_NFN_SCORE:
            errorMsgPtr = "The NFN score must be between 0.0 and
1.0";
    }
}

```

```

        break;
    case NH_INVALID_FNU_SCORE:
        errorMsgPtr = "The FNU score must be between 0.0 and
1.0";
        break;
    case NH_INVALID_NLN_SCORE:
        errorMsgPtr = "The NLN score must be between 0.0 and
1.0";
        break;
    case NH_INVALID_LNU_SCORE:
        errorMsgPtr = "The LNU score must be between 0.0 and
1.0";
        break;
    case NH_INVALID_GN_ANCHOR_FACTOR:
        errorMsgPtr = "The GN anchor score must be between 0.0
and 1.0";
        break;
    case NH_INVALID_NH_ANCHOR_FACTOR:
        errorMsgPtr = "The SN anchor score must be between 0.0
and 1.0";
        break;
    case NH_INVALID_GN_OOPS_FACTOR:
        errorMsgPtr = "The GN OOPS factor must be between 0.0
and 1.0";
        break;
    case NH_INVALID_NH_OOPS_FACTOR:
        errorMsgPtr = "The SN OOPS factor must be between 0.0
and 1.0";
        break;
    case NH_INVALID_ABS_DEL_GN_TAQ_FACTOR:
        errorMsgPtr = "The Abs delete GN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_ABS_DIS_GN_TAQ_FACTOR:
        errorMsgPtr = "The Abs disregard GN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_ABS_DEL_NH_TAQ_FACTOR:
        errorMsgPtr = "The Abs delete SN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_ABS_DIS_NH_TAQ_FACTOR:
        errorMsgPtr = "The Abs disregard SN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_DEL_GN_TAQ_FACTOR:
        errorMsgPtr = "The delete GN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_DIS_GN_TAQ_FACTOR:
        errorMsgPtr = "The disregard GN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_DEL_NH_TAQ_FACTOR:
        errorMsgPtr = "The delete SN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_DIS_NH_TAQ_FACTOR:
        errorMsgPtr = "The disregard SN TAQ factor must be
between 0.0 and 1.0";
        break;

```

```

        case NH_INVALID_GN_COMPRESSED_NAME_SCORE:
            errorMsgPtr = "The GN compressed name score must be
between 0.0 and 1.0";
            break;
        case NH_INVALID_NH_COMPRESSED_NAME_SCORE:
            errorMsgPtr = "The SN compressed name score must be
between 0.0 and 1.0";
            break;
        case NH_RESULTS_LIST_INSERT_ALLOC_FAILURE:
            errorMsgPtr = "Could not allocate space for a new
results list";
            break;
        case NH_GN_VAR_TABLE_CREATION_ERROR:
            errorMsgPtr = "Problem creating GN variant table";
            break;
        case NH_NH_VAR_TABLE_CREATION_ERROR:
            errorMsgPtr = "Problem creating SN variant table";
            break;
        case NH_TAQ_TABLE_CREATION_ERROR:
            errorMsgPtr = "Problem creating TAQ table";
            break;
        case NH_SEG_BREAK_CHARS_CREATION_ERROR:
            errorMsgPtr = "Problem creating segment break
characters string";
            break;
        case NH_NOISE_CHARS_CREATION_ERROR:
            errorMsgPtr = "Problem creating noise characters
string";
            break;
        case NH_INVALID_RESULTS_LIST_SIZE:
            errorMsgPtr = "Invalid size requested for results
list";
            break;
        case NH_RESULTS_LIST_ALLOCATION_ERROR:
            errorMsgPtr = "Problem creating internal results list
storage";
            break;
        case NH_RESULTS_ARRAY_NULL_ERROR:
            errorMsgPtr = "Internal results list storage is
invalid";
            break;
        case NH_TAQ_RECORD_ALLOC_ERROR:
            errorMsgPtr = "Problem allocating space for new TAQ
record";
            break;
        case NH_VARIANT_ALLOC_ERROR:
            errorMsgPtr = "Problem allocating space for new
variant record";
            break;
        case NH_VARIANTS_DONT_EXIST:
            errorMsgPtr = "The supplied names are not currently
variants";
            break;
        case NH_INVALID_VARIANT_SCORE:
            errorMsgPtr = "Variant scores must be between 0.0 and
1.0";
            break;
        case NH_MAX_VARIANT_SIZE_INCREMENT_FAILED:
            errorMsgPtr = "Could not increase variant storage to
add new variant relationship";
            break;

```

```

        case NH_VARIANT_ALREADY_RELATED:
            errorMsgPtr = "The names are already related to each
other";
            break;
        case NH_COMP_PARMS_BAD_STREAM_ON_CONSTRUCT:
            errorMsgPtr = "The comp parameters stream passed to
the constructor is invalid";
            break;
        case NH_COMP_PARMS_BAD_STREAM_ON_ARCHIVE:
            errorMsgPtr = "The comp parameters stream passed to
the archiveData method is invalid";
            break;
        case NH_NAME_PARMS_FILE_NOISE_CHARS_ERROR:
            errorMsgPtr = "The noise characters could not be
read";
            break;
        case NH_NAME_PARMS_FILE_BREAKS_CHARS_ERROR:
            errorMsgPtr = "The break characters could not be
read";
            break;
        case NH_NAME_PARMS_BAD_STREAM_ON_CONSTRUCT:
            errorMsgPtr = "The Name Parameters stream passed to
the constructor was bad";
            break;
        case NH_NAME_PARMS_BAD_STREAM_ON_WRITE:
            errorMsgPtr = "The Name Parameters stream passed to
the archive method was bad";
            break;
        case NH_NAME_PARMS_FILE_BAD_CULTURE_CODE:
            errorMsgPtr = "The culture code read from the Name
parameters stream was invalid";
            break;
        case NH_TAQ_NOT_FOUND:
            errorMsgPtr = "The specified TAQ could not be found";
            break;
        case NH_TAQ_ALREADY_EXISTS:
            errorMsgPtr = "The specified TAQ is already defined";
            break;
        case NH_INVALID_GN_THRESH:
            errorMsgPtr = "The GN Threshold must be between 0.0
and 1.0";
            break;
        case NH_INVALID_NH_THRESH:
            errorMsgPtr = "The SN Threshold must be between 0.0
and 1.0";
            break;
        case NH_INVALID_GN_WEIGHT:
            errorMsgPtr = "The GN Weight must be between 0.0 and
1.0";
            break;
        case NH_INVALID_NH_WEIGHT:
            errorMsgPtr = "The SN Weight must be between 0.0 and
1.0";
            break;
        case NH_INVALID_CULTURE_CODE:
            errorMsgPtr = "The specified culture code is invalid";
            break;
        case NH_ERROR_READING_CUSTOM_PARAMETER_FROM_FILE:
            errorMsgPtr = "A problem was encounter when reading a
custom parameter from a file";
            break;

```

```
        case NH_ERROR_WRITING_CUSTOM_PARAMETER_TO_FILE:
            errorMsgPtr = "A problem was encounter when writing a
custom parameter to a file";
            break;
        default:
            errorMsgPtr = "Unknown Error";
            break;
    }
    strncpy(textBuffer, errorMsgPtr, maxChars);
    textBuffer[maxChars] = EOS;
}
```

```

// File: NH_culture_codes.cpp
//
// Description:
//
//      Definition of global array of culture code strings
//
//
// History:
//
//      9/12/97      EFB      Created
//      3/20/98      EFB      Changed names to NH from SN
//

#include <string.h>

#include "NH_culture_codes.h"

// The following two global arrays must be the same size.
// That is, they must have the same number of elements.
// If you add or remove items, you must also update the
// constant NH_NUM_CULTURE_CODES
// In addition, they must maintain the same relative order
// (for example, Arabic must be in the same position in both
// arrays).
// lastly, this stuff must match the NHParmsType enum type,
// both in number and relative position. The NH_NUM_PARMS_TYPES
// must also be kept in sync as well.
char *NH_culture_codes[] = { NH_CULTURE_CODE_ANGLO,

                             NH_CULTURE_CODE_ARABIC,

                             NH_CULTURE_CODE_CHINESE,

                             NH_CULTURE_CODE_GENERIC,

                             NH_CULTURE_CODE_HISPANIC,

                             NH_CULTURE_CODE_KOREAN,

                             NH_CULTURE_CODE_RUSSIAN};

char *NH_culture_strings[] = { NH_CULTURE_STRING_ANGLO,

                                NH_CULTURE_STRING_ARABIC,

                                NH_CULTURE_STRING_CHINESE,

                                NH_CULTURE_STRING_GENERIC,

                                NH_CULTURE_STRING_HISPANIC,

                                NH_CULTURE_STRING_KOREAN,

                                NH_CULTURE_STRING_RUSSIAN};

bool NH_validate_culture_code(NHCultureCode cultureCode)
{
    bool found = false;

```



```
    for (int i = 0; i < NH_NUM_CULTURE_CODES; i++) {  
        if (!strncmp(cultureCode, NH_culture_codes[i],  
NH_MAX_CULTURE_CODE_LEN)) {  
            found = true;  
            break;  
        }  
    }  
    return found;  
}
```

```

// File: namehunter.h
//
// Description:
//
// shutdown and startup functions for the NameHunter system.
// These are really just blind interfaces to the
// NH_variant_taq_globals functions. We do this because
// we want to hide the details of the variants and TAQs
// from the API user.
//
//
// History:
//
//          9/9/97      EFB      Created
//          3/20/98     EFB      Changed names to NH from SN

#include "namehunter.h"
#include "NHVariantTable.hpp"
#include "NHQAQTable.hpp"
#include "NH_variant_taq_globals.h"
#include "NHDigraphBitmapArray.hpp"

extern NHVariantTable *NH_snVariantTable;
extern NHVariantTable *NH_gnVariantTable;
extern NHQAQTable *NH_taqTable;

NHDigraphBitmapArray globalDigraphBitmapArray;

void NH_startup()
{
    NH_getVariantTable(NH_SURNAME_VARIANTS);
    NH_getVariantTable(NH_GIVENNAME_VARIANTS);
    NH_getTAQTable();
}

void NH_shutdown()
{
    if (NH_snVariantTable != NULL) {
        delete NH_snVariantTable;
        NH_snVariantTable = NULL;
    }
    if (NH_gnVariantTable != NULL) {
        delete NH_gnVariantTable;
        NH_gnVariantTable = NULL;
    }
    if (NH_taqTable != NULL) {
        delete NH_taqTable;
        NH_taqTable = NULL;
    }
}

```

```
// File: NH_getErrorText.cpp
```

```
//
```

```
// Description:
```

```
//
```

```
// Implementation to the NH_getErrorText function. This function can  
// be used to return the error text for an associated error code.
```

```
//
```

```
//
```

```
// History:
```

```
//
```

```
// 6/23/97 EFB Created
```

```
// 3/20/98 EFB Changed names to NH from SN
```

```
//
```

```
#include "NH_get_error_text.h"
```

```
#include <string.h>
```

```
void NH_get_error_text(NHReturnCode errorCode, char *textBuffer, int maxChars)
```

```
{
```

```
    char *errorMsgPtr;
```

```
    switch (errorCode) {
```

```
        case NH_SUCCESS:
```

```
            errorMsgPtr = "Operation successful";
```

```
            break;
```

```
        case NH_MATCH:
```

```
            errorMsgPtr = "The comparison matched";
```

```
            break;
```

```
        case NH_NO_MATCH:
```

```
            errorMsgPtr = "The comparison did not match";
```

```
            break;
```

```
        case NH_INVALID_SCORE_THRESH:
```

```
            errorMsgPtr = "The threshold must be between 0.0 and 1.0";
```

```
            break;
```

```
        case NH_INVALID_GN_INIT_SCORE:
```

```
            errorMsgPtr = "The GN initial score must be between 0.0 and 1.0";
```

```
            break;
```

```
        case NH_INVALID_NH_INIT_SCORE:
```

```
            errorMsgPtr = "The SN initial score must be between 0.0 and 1.0";
```

```
            break;
```

```
        case NH_INVALID_GN_INIT_ON_INIT_MATCH_SCORE:
```

```

        errorMsgPtr = "The GN initial on initial match score must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_NH_INIT_ON_INIT_MATCH_SCORE:
        errorMsgPtr = "The SN initial on initial match score must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_NFN_SCORE:
        errorMsgPtr = "The NFN score must be between 0.0 and 1.0";
        break;
    case NH_INVALID_FNU_SCORE:
        errorMsgPtr = "The FNU score must be between 0.0 and 1.0";
        break;
    case NH_INVALID_NLN_SCORE:
        errorMsgPtr = "The NLN score must be between 0.0 and 1.0";
        break;
    case NH_INVALID_LNU_SCORE:
        errorMsgPtr = "The LNU score must be between 0.0 and 1.0";
        break;
    case NH_INVALID_GN_ANCHOR_FACTOR:
        errorMsgPtr = "The GN anchor score must be between 0.0 and
1.0";
        break;
    case NH_INVALID_NH_ANCHOR_FACTOR:
        errorMsgPtr = "The SN anchor score must be between 0.0 and
1.0";
        break;
    case NH_INVALID_GN_OOPS_FACTOR:
        errorMsgPtr = "The GN OOPS factor must be between 0.0 and
1.0";
        break;
    case NH_INVALID_NH_OOPS_FACTOR:
        errorMsgPtr = "The SN OOPS factor must be between 0.0 and
1.0";
        break;
    case NH_INVALID_ABS_DEL_GN_TAQ_FACTOR:
        errorMsgPtr = "The Abs delete GN TAQ factor must be between
0.0 and 1.0";
        break;
    case NH_INVALID_ABS_DIS_GN_TAQ_FACTOR:
        errorMsgPtr = "The Abs disregard GN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_ABS_DEL_NH_TAQ_FACTOR:

```

```

0.0 and 1.0";
        errorMsgPtr = "The Abs delete SN TAQ factor must be between
0.0 and 1.0";
        break;
    case NH_INVALID_ABS_DIS_NH_TAQ_FACTOR:
        errorMsgPtr = "The Abs disregard SN TAQ factor must be
between 0.0 and 1.0";
        break;
    case NH_INVALID_DEL_GN_TAQ_FACTOR:
        errorMsgPtr = "The delete GN TAQ factor must be between 0.0
and 1.0";
        break;
    case NH_INVALID_DIS_GN_TAQ_FACTOR:
        errorMsgPtr = "The disregard GN TAQ factor must be between 0.0
and 1.0";
        break;
    case NH_INVALID_DEL_NH_TAQ_FACTOR:
        errorMsgPtr = "The delete SN TAQ factor must be between 0.0
and 1.0";
        break;
    case NH_INVALID_DIS_NH_TAQ_FACTOR:
        errorMsgPtr = "The disregard SN TAQ factor must be between 0.0
and 1.0";
        break;
    case NH_INVALID_GN_COMPRESSED_NAME_SCORE:
        errorMsgPtr = "The GN compressed name score must be between
0.0 and 1.0";
        break;
    case NH_INVALID_NH_COMPRESSED_NAME_SCORE:
        errorMsgPtr = "The SN compressed name score must be between
0.0 and 1.0";
        break;
    case NH_RESULTS_LIST_INSERT_ALLOC_FAILURE:
        errorMsgPtr = "Could not allocate space for a new results list";
        break;
    case NH_GN_VAR_TABLE_CREATION_ERROR:
        errorMsgPtr = "Problem creating GN variant table";
        break;
    case NH_NH_VAR_TABLE_CREATION_ERROR:
        errorMsgPtr = "Problem creating SN variant table";
        break;
    case NH_TAQ_TABLE_CREATION_ERROR:
        errorMsgPtr = "Problem creating TAQ table";
        break;
    case NH_SEG_BREAK_CHARS_CREATION_ERROR:
        errorMsgPtr = "Problem creating segment break characters string";

```

```

        break;
    case NH_NOISE_CHARS_CREATION_ERROR:
        errorMsgPtr = "Problem creating noise characters string";
        break;
    case NH_INVALID_RESULTS_LIST_SIZE:
        errorMsgPtr = "Invalid size requested for results list";
        break;
    case NH_RESULTS_LIST_ALLOCATION_ERROR:
        errorMsgPtr = "Problem creating internal results list storage";
        break;
    case NH_RESULTS_ARRAY_NULL_ERROR:
        errorMsgPtr = "Internal results list storage is invalid";
        break;
    case NH_TAQ_RECORD_ALLOC_ERROR:
        errorMsgPtr = "Problem allocating space for new TAQ record";
        break;
    case NH_VARIANT_ALLOC_ERROR:
        errorMsgPtr = "Problem allocating space for new variant record";
        break;
    case NH_VARIANTS_DONT_EXIST:
        errorMsgPtr = "The supplied names are not currently variants";
        break;
    case NH_INVALID_VARIANT_SCORE:
        errorMsgPtr = "Variant scores must be between 0.0 and 1.0";
        break;
    case NH_MAX_VARIANT_SIZE_INCREMENT_FAILED:
        errorMsgPtr = "Could not increase variant storage to add new
variant relationship";
        break;
    case NH_VARIANT_ALREADY_RELATED:
        errorMsgPtr = "The names are already related to each other";
        break;
    case NH_COMP_PARMS_BAD_STREAM_ON_CONSTRUCT:
        errorMsgPtr = "The comp parameters stream passed to the
constructor is invalid";
        break;
    case NH_COMP_PARMS_BAD_STREAM_ON_ARCHIVE:
        errorMsgPtr = "The comp parameters stream passed to the
archiveData method is invalid";
        break;
    case NH_NAME_PARMS_FILE_NOISE_CHARS_ERROR:
        errorMsgPtr = "The noise characters could not be read";
        break;
    case NH_NAME_PARMS_FILE_BREAKS_CHARS_ERROR:
        errorMsgPtr = "The break characters could not be read";

```

```

        break;
    case NH_NAME_PARMS_BAD_STREAM_ON_CONSTRUCT:
        errorMsgPtr = "The Name Parameters stream passed to the
constructor was bad";
        break;
    case NH_NAME_PARMS_BAD_STREAM_ON_WRITE:
        errorMsgPtr = "The Name Parameters stream passed to the archive
method was bad";
        break;
    case NH_NAME_PARMS_FILE_BAD_CULTURE_CODE:
        errorMsgPtr = "The culture code read from the Name parameters
stream was invalid";
        break;
    case NH_TAQ_NOT_FOUND:
        errorMsgPtr = "The specified TAQ could not be found";
        break;
    case NH_TAQ_ALREADY_EXISTS:
        errorMsgPtr = "The specified TAQ is already defined";
        break;
    case NH_INVALID_GN_THRESH:
        errorMsgPtr = "The GN Threshold must be between 0.0 and 1.0";
        break;
    case NH_INVALID_NH_THRESH:
        errorMsgPtr = "The SN Threshold must be between 0.0 and 1.0";
        break;
    case NH_INVALID_GN_WEIGHT:
        errorMsgPtr = "The GN Weight must be between 0.0 and 1.0";
        break;
    case NH_INVALID_NH_WEIGHT:
        errorMsgPtr = "The SN Weight must be between 0.0 and 1.0";
        break;
    case NH_INVALID_CULTURE_CODE:
        errorMsgPtr = "The specified culture code is invalid";
        break;
    case
NH_ERROR_READING_CUSTOM_PARAMETER_FROM_FILE:
        errorMsgPtr = "A problem was encounter when reading a custom
parameter from a file";
        break;
    case NH_ERROR_WRITING_CUSTOM_PARAMETER_TO_FILE:
        errorMsgPtr = "A problem was encounter when writing a custom
parameter to a file";
        break;
    default:
        errorMsgPtr = "Unknown Error";

```

```
        break;
    }
    strncpy(textBuffer, errorMsgPtr, maxChars);
    textBuffer[maxChars] = EOS;
}
```



```

//      File: namehunter.h
//
//      Description:
//
//          shutdown and startup functions for the NameHunter system.
//          These are really just blind interfaces to the
//          NH_variant_taq_globals functions. We do this because
//          we want to hide the details of the variants and TAQs
//          from the API user.
//
//
//      History:
//
//          9/9/97 EFB          Created
//          3/20/98      EFB      Changed names to NH from SN

#include      "namehunter.h"
#include      "NHVariantTable.hpp"
#include      "NHTAQTable.hpp"
#include      "NH_variant_taq_globals.h"
#include      "NHDigraphBitmapArray.hpp"


extern NHVariantTable      *NH_snVariantTable;
extern NHVariantTable      *NH_gnVariantTable;
extern NHTAQTable          *NH_taqTable;


NHDigraphBitmapArray      globalDigraphBitmapArray;


void  NH_startup()
{
    NH_getVariantTable(NH_SURNAME_VARIANTS);
    NH_getVariantTable(NH_GIVENNAME_VARIANTS);
    NH_getTAQTable();
}


void  NH_shutdown()
{
    if (NH_snVariantTable != NULL)  {
        delete NH_snVariantTable;
        NH_snVariantTable = NULL;
    }
}

```

```
if (NH_gnVariantTable != NULL) {  
    delete NH_gnVariantTable;  
    NH_gnVariantTable = NULL;  
}  
if (NH_taqTable != NULL) {  
    delete NH_taqTable;  
    NH_taqTable = NULL;  
}  
}
```

```

// File: NHVariantTable.hpp
//
// Description:
//
// Interface to the NHVariantTable class.
//
//
// History:
//
// 5/7/97 EFB Created
// 6/23/97 EFB Changed processing to get rid of
variant types as assign an
// individual score for each variant pair.
// 6/23/97 EFB Enhanced comments.
// 9/9/97 EFB Added support for a culture code in
the variant object,
// which required
changes to this object's interaction
// with the NHVariant
class.
// 3/20/98 EFB Changed names to NH from SN
//

```

/\*

Variant information consists of two names that are related, along with a designation of variant type, which describes how the two names are related.

The following holds true in our model:

- if Name A is related to name B with varType V, then B is related to A with varType V.
  - When constructing the table, only one of the pairs (A, B) or (B, A) should be entered.
- The internals will ensure that a request of "is B related to A" and a request of "is A related to B" will work.
- Name variants are single segments.

Internally, we represent the information as a hash table of NH\_VarHashTableRecord structures. Each of these structures contains a name string, plus a Variant object. Each Variant object (a separate class) has the following:

```

NHVarId id;
// unique id for each variant
byte numRelatedVariants; // number of
other variants we are related to
NHVarId variants[MAX_VARIANTS_PER_NAME]
// array of id's
double varScores[MAX_VARIANTS_PER_NAME] //
score for each variant

// as related to this variant

```

```

        short int          varCultures[MAX_VARIANTS_PER_NAME] // score
for each variant

```

```

// as related to this variant

```

The name of the variant is actually stored in the hash table node, rather than the variant object.

There are three important functions in the VariantTable class:

```

        bool          addVariant(char *name1, char *name2,
NHVarType varType, char *cultCode);
        NHVariant      getVariantObjectName(char *name);
        NHVarId        getVariantIdForName(char *name);

```

```

        // The Variant has the method:
        double          getVariantScoreForIdAndCulture(NHVarId varId,
char *cultureCode);

```

The variant table is built by multiple calls to addVariant() from the constructor. There is one call to addVariant() for each pair of names that are related.

addVariant() takes 2 names that are related, along with a culture code to describe the relationship.

getVariantInfoForName returns the NHVariant object associated with the name (or NULL).

getVariantIdForName() returns the id associated with the name.

Typically, a QueryNameData object gets a pointer to it's variant object up front. Each time it gets compared to an EvalNameData object, it calls the getVariantIdForName() method to get an id, which it then passes the to the getVariantScoreForId() to see if the two are related.

```

*/

```

```

#ifndef      NHVARIANTTABLE_HPP
#define NHVARIANTTABLE_HPP

```

```

#include      "NHVariant.hpp"
#include      "NH_get_error_text.h"

```

```

// define a const for end of string
#ifndef      EOS

```

```

#define      EOS      '\0'
#endif

//      how long can a variant be ?
#define      NH_MAX_VARIANT_LEN      30

//      define a type to specify the type of variant table
//      types are defined by a combination of culture and
//      name field.
enum NH_VARIANT_TABLE_TYPES
{
    NH_SURNAME_VARIANTS,
    NH_GIVENNAME_VARIANTS,
    NH_EMPTY_VARIANTS
};

//      define a record in the Variant hash table
typedef struct NH_VAR_HASH_TABLE_RECORD_T {
    char
        segment[NH_MAX_VARIANT_LEN + 1];
    NHVariant
        *variant;
    struct
        NH_VAR_HASH_TABLE_RECORD_T      *next;
} NH_VarHashTableRecord;
//      pointer to
//      next node in hash chain
} NH_VarHashTableRecord;

//      Do not change without seeing member function hash().
#define NH_MAX_VAR_HASH_TABLE_NODES 907

//      define a type that is a pointer to a NH_VarTableRecord
typedef NH_VarHashTableRecord *NH_VarHashTableRecordPtr;

//      define a type that is a table (array) of NH_VarTableRecord
typedef NH_VarHashTableRecordPtr
NH_VariantHashTable[NH_MAX_VAR_HASH_TABLE_NODES];

class NHVariantTable
{
public:
    NHVariantTable(NH_VARIANT_TABLE_TYPES tableType);
    virtual ~NHVariantTable();

    //      returns the NHVariant object associated with the name,
    //      or NULL is there is no object for the name.
    NHVariant      *      getVariantObjectForName(char *name);

    //      returns the NHVarId associated with the name.  If
    //      there is
    //      no variant for the name, the function returns
    NH_VAR_NOT_FOUND.
    NHVarId      getVariantIdForName(char *name);

    NHReturnCode      getStatus() {return
status;}

```

```

        NHReturnCode      addVariant(char *name1,
char *name2, double varScore, char *cultCode);

        int      getNumHashBuckets()      {return
NH_MAX_VAR_HASH_TABLE_NODES;}

        NH_VarHashTableRecordPtr      getHashBucketStartNodeAt(int
hashTableIndex)

                {return variantHashTable[hashTableIndex];}

        //      function to change the score associated with two
variants with a
        //      specified culture.
        //      The function return:
        //
        //      NH_SUCCESS - if things worked out OK
        //      NH_VARIANTS_DONT_EXIST - if the either name does
not exist in the table
        //
        //      or the names are not already variants of
each
        //
        //      other with the specified culture.
        //      NH_INVALID_VARIANT_SCORE - if the score is
invalid
        NHReturnCode      changeVariantScore(char *name1, char
*name2, char *cultureCode, double newScore);

        //      a function to remove the relationship between two
variants within
        //      a specified culture.
        //      This function is used for the VariantManager
application.
        //      If either variant ends up without a relationship after
this
        //      operation, it is left in, but when saved, the
resulting file
        //      will contain a "*" rather than a related name. The
function can
        //      return
        //
        //      NH_SUCCESS - if things worked out OK
        //      NH_VARIANTS_DONT_EXIST - if the names are not
already variants
        NHReturnCode      removeVariantRelation(char *name1, char
*name2, char *cultureCode);

        //      return the next available id, which is the number of
//      distinct variants in our table.
        NHVarId      getNextAvailableVarId() {return
nextAvailableVarId;}

        bool      getDirty() {return dirty;}
        void      setDirty(bool aBool)      {dirty = aBool;}

protected:

        //      add a variant relationship.
        virtual      NHVariant      *      getOrCreateVariantObjectForNam

```

```

e(char *name);

    NHVarId          nextAvailableVarId;

    NH_VariantHashTable    variantHashTable;

    NHReturnCode          status;          //    are we
valid

    bool                dirty;          //    have we changed

    //    Returns an integer in the range [0,
NH_MAX_VAR_HASH_TABLE_NODES].
    inline unsigned int NHVariantTable::hash(char *string)
    {
        char
        unsigned    int        i;        *p;
        unsigned    int        sum;

        for (p = string, i = 2, sum = 0; *p != EOS; p++, i +=
2)
            sum += i * *p;
        return sum % NH_MAX_VAR_HASH_TABLE_NODES;
    } // hash

private:

};

#endif

```

```

// File: NHVariantTable.cpp
//
// Description:
//
//      Implementation to the NHVariantTable class.
//
//
// History:
//
//      5/14/97      EFB      Created
//      3/20/98      EFB      Changed names to NH from SN
//

#include <string.h>
#include <stdio.h>

#include "NHVariantTable.hpp"
#include "NH_util.hpp"
#include "NH_culture_codes.h"

NHVariantTable::NHVariantTable(NH_VARIANT_TABLE_TYPES tableType)
{
    status = NH_SUCCESS;
    dirty = false;
    // clear out the hash table
    for (int i = 0; i < NH_MAX_VAR_HASH_TABLE_NODES; i++)
        variantHashTable[i] = NULL;

    // initialize our variant id variable.
    nextAvailableVarId = 0;

    /* gnv test stuff
        addVariant("ED", "EDWARD", 0.7, "E ");
        addVariant("GERRY", "GENERIC", 0.7, "G ");
        addVariant("HOP", "HOPSING", 0.7, "C ");
        addVariant("NASSIR", "NARADMAN", 0.7, "A ");
        addVariant("BORRIS", "NATASIA", 0.7, "R ");
        addVariant("JUAN", "EPSTEIN", 0.7, "H ");
        addVariant("KORY", "KOREAN", 0.7, "H ");
    */

    /* snv test stuff
        addVariant("HUANG", "WONG", 0.7, "C ");
    */

    // the following include lines are commented out because it
    takes forever
    // to compile release versions when they are left in.
    if (tableType == NH_GIVENNAME_VARIANTS) {
        // #include "gnvdata.h"
    }
    else if (tableType == NH_SURNAME_VARIANTS) {
        // #include "snvdata.h"
    }
}

```



```

//    release all the memory used to store NH_VarHashTableRecord
pointers
NHVariantTable::~NHVariantTable()
{
    NH_VarHashTableRecordPtr    prevRecord;
    NH_VarHashTableRecordPtr    varRecord;
    unsigned int                tableIndex;

    for (tableIndex = 0; tableIndex < NH_MAX_VAR_HASH_TABLE_NODES;
tableIndex++)    {
        varRecord = variantHashTable[tableIndex];
        while (varRecord != NULL)    {
            prevRecord = varRecord;
            varRecord = varRecord->next;
            //    delete the record we allocated,
            //    as well as the SNVariant object pointed to by
the
            //    variant member of this record
            delete prevRecord->variant;
            delete prevRecord;
        }
    }
}

//    returns the NHVariant object associated with the name,
//    or NULL if there is no object for the name.
NHVariant *    NHVariantTable::getVariantObjectForName(char *name)
{
    NHVariant *variantObject = NULL;
    unsigned int    tableIndex;
    NH_VarHashTableRecordPtr    tempRecordPtr;

    //    find the hash value for the (possible) variant
    tableIndex = hash(name);

    //    go through the records in the chain at that offset in the
    //    hash table, and try to find the variant we are looking for.
    tempRecordPtr = variantHashTable[tableIndex];
    while (tempRecordPtr != NULL)    {
        if (!strcmp(tempRecordPtr->segment, name))    {
            variantObject = tempRecordPtr->variant;
            break;
        }
        else    //    move on to next record in the chain
            tempRecordPtr = tempRecordPtr->next;
    }
    return variantObject;
}

//    returns the NHVariant object associated with the name,
//    or creates a new one.
NHVariant *    NHVariantTable::getOrCreateVariantObjectForName(char
*name)
{
    NHVariant *variantObject = getVariantObjectForName(name);

```

```

if (variantObject == NULL) {
    // no object existed before, so create one and add it
    // to the hash table.

    unsigned
int      tableIndex;
        NH_VarHashTableRecordPtr prevRecord;
        NH_VarHashTableRecordPtr newVariantHashTableRecord =
new NH_VarHashTableRecord;

    variantObject = new NHVariant(nextAvailableVarId++);
    if (variantObject != NULL) {
        // find the hash value for the name
        tableIndex = hash(name);

        // fill up the values in the record
        strncpy(newVariantHashTableRecord->segment, name,
NH_MAX_VARIANT_LEN);
        newVariantHashTableRecord->segment[NH_MAX_VARIANT_LEN]
= EOS;

        newVariantHashTableRecord->variant = variantObject;
        newVariantHashTableRecord->next = NULL;

        // now add the new record to the chain of entries
        // at that index.
        prevRecord = variantHashTable[tableIndex];
        if (prevRecord == NULL)
            variantHashTable[tableIndex] =
newVariantHashTableRecord;
        else {
            while (prevRecord->next != NULL) {
                prevRecord = prevRecord->next;
            }
            prevRecord->next = newVariantHashTableRecord;
        }
    }
    else
        status = NH_VARIANT_ALLOC_ERROR;
}
return variantObject;
}

// returns the NHVarId associated with the name. If there is
// no variant for the name, the function returns NH_VAR_NOT_FOUND.
NHVarId NHVariantTable::getVariantIdForName(Char *name)
{
    NHVariant *variantObject = getVariantObjectForName(name);
    NHVarId returnId;

    if (variantObject != NULL) {
        returnId = variantObject->getVariantId();
    }
    else
        returnId = NH_VAR_NOT_FOUND;

    return returnId;
}

```

```

//      Add a variant relationship.
//      In order to do this, we must:
//
//      -      make sure both names already have entries in the hash
table
//      and if not, create them.
//      -      get the id of each entry.
//      -      add the id of each item to the variant information of
the other.
//
//      We handle the special case where the second name is a *. This
means
//      that the name should be part of the variant table, but not related
//      to anything. In this case,
//      we only create (or get) a NHVariant object for the name.
NHReturnCode      NHVariantTable::addVariant(char *name1, char *name2,
double varScore,
                                                    char *cultureCode)
{
    NHReturnCode      rc = NH_SUCCESS;
    NHVariant          *varObject1;
    NHVariant          *varObject2;

    if ((varScore < 0.0) || (varScore > 1.0))
        rc = NH_INVALID_VARIANT_SCORE;
    else {
        if (NH_validate_culture_code(cultureCode)) {
            //      Get variant object for both names. This will
also create
            //      a new entry if the name(s) were not in the table
already
            varObject1 = getOrCreateVariantObjectForName(name1);

            //      if the second name was a *, skip the creation of
the second
            //      NHVariant object and do not associate the names.
            if (strcmp(name2, "*") {
                varObject2 =
getOrCreateVariantObjectForName(name2);

                if ((varObject1 != NULL) && (varObject2 !=
NULL)) {
                    //      now associate each with the other,
using the supplied variant type
                    rc = varObject1->addVariant(varObject2,
cultureCode, varScore);
                    if (rc == NH_SUCCESS)
                        rc = varObject2->
addVariant(varObject1, cultureCode, varScore);
                }
            }
            else {
                //      flag it as an error, but do not mark the entire
table as bad
                rc = NH_INVALID_CULTURE_CODE;
            }
        }
    }
}

```

```

    }

    return rc;
}

// function to change the score associated with two variants.
// The function return:
//
//      NH_SUCCESS - if things worked out OK
//      NH_VARIANTS_DONT_EXIST - if the either name does not exist
//      in the table
//
//      or the names are not already variants of each
//
//      other
//      NH_INVALID_VARIANT_SCORE - if the score is invalid
NHReturnCode NHVariantTable::changeVariantScore(char *name1, char
*name2, char *cultureCode, double newScore)
{
    NHReturnCode rc = NH_SUCCESS;

    if ((newScore < 0.0) || (newScore > 1.0))
        rc = NH_INVALID_VARIANT_SCORE;
    else {
        NHVariant *var1 = getVariantObjectForName(name1);
        NHVariant *var2 = getVariantObjectForName(name2);

        if ((var1 == NULL) || (var2 == NULL))
            rc = NH_VARIANTS_DONT_EXIST;
        else {
            rc = var1->setVariantScoreForIdAndCulture(var2-
>getVariantId(), cultureCode, newScore);
            if (rc == NH_SUCCESS)
                rc = var2->setVariantScoreForIdAndCulture(var1-
>getVariantId(), cultureCode, newScore);
            // we should never have a case where the
items are related
            // in one direction but not the other.
        }
    }

    return rc;
}

// a function to remove the relationship between two variants.
// If either variant ends up without a relationship after this
// operation, it is left in, but when saved, the resulting file
// will contain a "*" rather than a related name. The function can
// return
//
//      NH_SUCCESS - if things worked out OK
//      NH_VARIANTS_DONT_EXIST - if the names are not already
variants
NHReturnCode NHVariantTable::removeVariantRelation(char *name1,
char *name2, char *cultureCode)
{
    NHReturnCode rc = NH_VARIANTS_DONT_EXIST;
    NHVariant *var1 = getVariantObjectForName(name1);

```

```

        NHVariant          *var2 = getVariantObjectForName(name2);

        if ((var1 == NULL) || (var2 == NULL))
            rc = NH_VARIANTS_DONT_EXIST;
        else {
            if (var1->removeVariant(var2->getVariantId(), cultureCode)
== NH_SUCCESS)
            {
                //      we should never have a case where the items are
related
                //      in one direction but not the other.
                if (var2->removeVariant(var1->getVariantId(),
cultureCode) == NH_SUCCESS)
                    rc = NH_SUCCESS;
            }
        }

        return rc;
    }
}

```

```

// File: NHVariant.hpp
//
// Description:
//
//         Interface to the NHVariant class.
//
//
// History:
//
//         6/6/97      EFB      Created
//         6/23/97     EFB      Changed processing to get rid of
variant types
//                                     as assign an
individual score for each variant pair.
//         9/9/97      EFB      Changed object so that each
relationship has an
//                                     associated
culture. Several access methods have
//                                     been changed to
allow for a culture specifier.
//         3/20/98     EFB      Changed names to NH from SN
//
//
/*
    Variant represents the variant information for one name.

    Currently, the name must be a single segment.

    The object contains the following information:

        NHVarId          id;
//                                     unique id
for this variant
    byte                numRelatedVariants;
//                                     how many variants are we related to?
        NHVarId          variantIds[MAX_VARIANTS_PER_NAME]; //
what are the id's of our related variants
    double              varScores[MAX_VARIANTS_PER_NAME]; //
Score for each variant

//    in variants array above
    short int           varCultures[MAX_VARIANTS_PER_NAME]; //    Two
byte code describing the culture

//    for this variant relationship. These are

//    actually char[2] codes.

    A variant knows how to add an id, type combination to its
information.
*/

#ifndef NHVARIANT_HPP
#define NHVARIANT_HPP

#include <stdlib.h>

```

```

#include "NH_get_error_text.h"
#include "NH_culture_codes.h"

typedef unsigned char byte;

// #define MAX_VARIANTS_PER_NAME 30
#define NH_INIT_VARIANTS_PER_NAME 5

// define a constant to represent that two variants were
// not related.
#define NH_VARIANTS_NOT_RELATED -1.0

// define a variant id as a short int.
typedef short int NHVarId;

#define NH_VAR_NOT_FOUND -1

// define a structure to hold the info about a related variant.
// We
// will use arrays of this structure to list the names related to
// a variant.
typedef struct NH_RELATED_VARIANTS_T {
    NHVarId variantId; // what is the id of our
    related variant
    double varScore; // Score for this
    variant, as related to the main variant

    // in variants array above
    char varCulture[NH_MAX_CULTURE_CODE_LEN];
    // Two byte code describing the culture

    // for this variant relationship. These are

    // actually char[2] codes.
} NH_RelatedVariants;

class NHVariant
{
public:
    NHVariant(NHVarId newId);
    virtual ~NHVariant();

    // Returns the variant score for the relationship between
    the
    // the supplied variant id and the variant, within the
    specified
    // culture. If the variants are not related, the
    function returns

```

```

        //      NH_VARIANTS_NOT_RELATED.
        double getVariantScoreForIdAndCulture(NHVarId relatedVarId,
char *cultCode);

        //      allows caller to search for across cultures within
this variant
        double getVariantScoreForIdAndAnyCulture(NHVarId
relatedVarId, char *cultCode);
        //      see if the supplied variant is related to us, and if
so,
        //      replace the existing score with the new score.
        //      if not, return NH_VARIANTS_DONT_EXIST.
        NHReturnCode setVariantScoreForIdAndCulture(NHVarId
relatedVarId,

char *cultCode, double score);

        //      adds the id of the specified variant (along with an
associated
        //      score and culture code) to our array of variants
related to us.
        virtual NHReturnCode addVariant(NHVariant *variant,
char *cultureCode,

double relatedVarScore);

        //      remove a variant from our list
        //      return NH_VARIANTS_DONT_EXIST if the id is not in our
list already
        virtual NHReturnCode removeVariant(NHVarId relatedVarId,
char *cultureCode);

        //      return the variant id for this object
        NHVarId getVariantId() {return id;}

        //      return the variant id for this object
        byte getNumVariants() {return numRelatedVariants;}

        NHVarId getIdForRelatedVariant(int relVarIndex)
        {
            NHVarId varId = 0;

            if ((relVarIndex > -1) && (relVarIndex <
numRelatedVariants))
                varId = relatedVariants[relVarIndex].variantId;
            return varId;
        }

        char * getCultureCodeForRelatedVariant(int relVarIndex)
        {
            char *cultureCode = NULL;

            if ((relVarIndex > -1) && (relVarIndex <
numRelatedVariants))
                cultureCode =
relatedVariants[relVarIndex].varCulture;
            return cultureCode;
        }

```



```

    },

    double      getScoreForRelatedVariant(int relVarIndex)
    {
        double      score = 0.0;

        if ((relVarIndex > -1) && (relVarIndex <
numRelatedVariants))
            score = relatedVariants[relVarIndex].varScore;
        return score;
    }

protected:
    NHVarId      id;
                // unique id
    for this variant
    byte      numRelatedVa
riants;      // how many variants are we related to?
    byte      maxRelatedVa
riants;      // how many variants are we related to?
    NH_RelatedVariants      *relatedVariants;

private:

};

#endif

```

```

//      File:  NHVariant.cpp
//
//      Description:
//
//      Implementation to the NHVariant class.
//
//
//      History:
//
//      6/6/97      EFB      Created
//      3/20/98     EFB      Changed names to NH from SN
//

```

```

#include <string.h>
#include <stdio.h>

```

```

#include      "NHVariant.hpp"
#include      "NH_util.hpp"

```

```

#ifndef      false
#define      false 0
#endif

```

```

#ifndef      true
#define      true 1
#endif

```

```

NHVariant::NHVariant(NHVarId newId)
{
    id = newId;
    numRelatedVariants = 0;
    maxRelatedVariants = NH_INIT_VARIANTS_PER_NAME;
    relatedVariants = new NH_RelatedVariants[maxRelatedVariants];
}

```

```

NHVariant::~~NHVariant()
{
    if (relatedVariants)
        delete [] relatedVariants;
}

```

```

//      see if the supplied variant is related to us, and if so, return
//      its score.
double      NHVariant::getVariantScoreForIdAndCulture(NHVarId
relatedVarId, char *cultCode)
{
    double      returnScore = NH_VARIANTS_NOT_RELATED;

    for (int i = 0; i < numRelatedVariants; i++)    {
        if ((relatedVariants[i].variantId == relatedVarId) &&
(memcmp(relatedVariants[i].varCulture, cultCode,
NH_MAX_CULTURE_CODE_LEN) == 0))    {
            returnScore = relatedVariants[i].varScore;
        }
    }
}

```

```

        break;
    }
    return returnScore;
}

// See if the supplied variant is related to us under any culture.
// Because this method is intended to be called several times (for
// possibly multiple cultures, it also takes a culture string that
// is used to keep track of the last culture that was returned. The
// first time the function is called, the culture is specified as an
// empty string. On return, it contains the first culture found
// in the list for the id. The next time the function is called,
// we look past that culture/id combination in the array looking for
// the next one, until we return NH_VARIANTS_NOT_RELATED:
double NHVariant::getVariantScoreForIdAndAnyCulture(NHVarId
relatedVarId, char *cultCode)
{
    double returnScore = NH_VARIANTS_NOT_RELATED;
    bool alreadyFoundLastCultCode = false;

    for (int i = 0; i < numRelatedVariants; i++) {
        if ((relatedVariants[i].variantId == relatedVarId)) {
            // ids matched, so see if they specified a culture
code
            if (*cultCode == EOS) {
                // this is first time through, so no check is
necessary.
                // copy the cult code into the supplied
string.
                NH_safe_strcpy(cultCode,
relatedVariants[i].varCulture, NH_MAX_CULTURE_CODE_LEN);
                returnScore = relatedVariants[i].varScore;
                break;
            }
            else {
                // this is not first time through, they are
passing us the cult code
                // that was found last time, so see if we
have already found that one
                if (alreadyFoundLastCultCode == true) {
                    NH_safe_strcpy(cultCode,
relatedVariants[i].varCulture, NH_MAX_CULTURE_CODE_LEN);
                    returnScore = relatedVariants[i].varScore;
                    break;
                }
                else {
                    // see if this is the cult code they
passed us
                    if (memcmp(relatedVariants[i].varCulture,
cultCode, NH_MAX_CULTURE_CODE_LEN) == 0) {
                        alreadyFoundLastCultCode =
true; // we found it
                    }
                }
            }
        }
    }
    return returnScore;
}

```



```

s) * maxRelatedVariants);
        delete [] relatedVariants;
        relatedVariants = biggerBlock;
        maxRelatedVariants *= 2;
    }
    else
        rc = NH_MAX_VARIANT_SIZE_INCREMENT_FAILED;
}

    if (rc == NH_SUCCESS) {
        relatedVariants[numRelatedVariants].variantId =
relatedVarId;
        relatedVariants[numRelatedVariants].varScore =
relatedVarScore;
        strncpy(relatedVariants[numRelatedVariants].varCulture,
cultureCode, NH_MAX_CULTURE_CODE_LEN);
        numRelatedVariants++;
    }

    return rc;
}

//    remove a variant from our list
//    return NH_VARIANTS_DONT_EXIST if the id is not in our list already
NHReturnCode    NHVariant::removeVariant(NHVarId relatedVarId, char
*cultureCode)
{
    NHReturnCode    rc = NH_VARIANTS_DONT_EXIST;

    for (int i = 0; i < numRelatedVariants; i++) {
        if ((relatedVariants[i].variantId == relatedVarId) &&
            (memcmp(relatedVariants[i].varCulture,
cultureCode, NH_MAX_CULTURE_CODE_LEN) == 0)) {
            //    now move any ids past the one that match
            //    back one space.
            for (int j = i + 1; j < numRelatedVariants;
j++) {
                relatedVariants[j - 1].varScore =
relatedVariants[j].varScore;
                relatedVariants[j - 1].variantId =
relatedVariants[j].variantId;
                strncpy(relatedVariants[j - 1].varCulture,
relatedVariants[j].varCu
lture, NH_MAX_CULTURE_CODE_LEN);
            }
            numRelatedVariants--;
            //    we not have one
            less variant
            rc = NH_SUCCESS;
            break;
        }
    }
    return rc;
}

```

```

// File: NHTAQTable.hpp
//
// Description:
//
// Interface to the NHTAQTable class.
//
//
// History:
//
// 5/7/97 EFB Created
// 3/20/98 EFB Changed names to NH from SN
//
//
// The TAQTable is organized by name and culture. That is the unique
key
// in the table. We do lookups by hashing the name, but must
consider the
// culture code as we walk the hash table bucket.

#ifndef NHTAQTABLE_HPP
#define NHTAQTABLE_HPP

#include "NH_culture_codes.h"
#include "NHNameData.hpp"
#include "NH_get_error_text.h"

// how many characters can a TAQ value be?
#define NH_MAX_TAQ_LEN 20

// define the possible values for the TAQ action
#define NH_TAQ_ACTION_DELETE 'X'
#define NH_TAQ_ACTION_DISREGARD 'D'

// define a record in the hash table of TAQ values
typedef struct NH_TAQ_RECORD_T {
    char taqString[NH_MAX_TAQ_LEN + 1]; // string that is the
TAQ value
    char taqType; // P, S, I, T or Q
    char sepIfConjoined; // Y or N
    char gnAction; // what to do when
found in gn
    char snAction; // what to do when
found in sn
    char taqCulture[NH_MAX_CULTURE_CODE_LEN +
1]; // which culture does this apply to?
0 struct NH_TAQ_RECORD_T *next; // pointer to next TAQ
record in this hash branch
} NH_TAQRecord;

// Do not change without seeing function NH_TAQhash().
#define NH_MAX_TAQ_HASH_NODES 907

// define a type that is a pointer to a NH_TAQRecord
typedef NH_TAQRecord *NH_TAQRecordPtr;

```

```

//    define a type that is a table (array) of NH_TAQRecordPtrs
typedef NH_TAQRecordPtr NH_TAQHashTable[NH_MAX_TAQ_HASH_NODES];

enum NH_TAQ_TABLE_TYPE {
    NH_PRODUCTION_TAQ_TABLE,
    NH_EMPTY_TAQ_TABLE
};

class NHTAQTable
{
public:
    NHTAQTable(NH_TAQ_TABLE_TYPE type);
    ~NHTAQTable();

    //    function to return a pointer to the TAQ structure for
the
    //    supplied character string (segment), cultureCode
combination.
    //    Returns NULL if the supplied segment is not known to
the TAQ table
    //    for the specified culture code.
    NH_TAQRecordPtr      getTAQSegment(char *nameSeg,
char *cultureCode);

    //    specialized version of the above function that looks
for the
    //    name segment in either of the specified culture codes.
It makes
    //    sure that if the name is found in the
primaryCultureCode, that one
    //    gets returned even if we come upon the
secondaryCultureCode first.
    NH_TAQRecordPtr      getTAQSegment(char *nameSeg,
char *primaryCultureCode,

                                char *secondaryCultureCode);

    NHReturnCode          getStatus() {return
status;}

    bool                  getDirty()
{return dirty;}
    void                  setDirty(boo
l aBool)    {dirty = aBool;}

    int                   getNum
HashBuckets()    {return NH_MAX_TAQ_HASH_NODES;}

    NH_TAQRecordPtr      getHashBucketStartNodeAt(int
hashTableIndex)

                                {return taqHashTable[hashTableIndex];}

    NHReturnCode          addTAQValue(char
*taqValue, char taqType,

                                char sepIfConjoined, char
gnTAQAction,

```

```

char snTAQAction, char *taqCulture);

NHReturnCode removeTAQValue(char
*taqValue, char *cultureCode);

protected:

private:
    // Returns an integer in the range [0,
NH_MAX_TAO_HASH_NODES].
    inline unsigned int hash(char *string)
    {
        char *p;
        unsigned int i;
        unsigned int sum;

        for (p = string, i = 2, sum = 0; *p != EOS; p++, i +=
2)
            sum += i * *p;
        return sum % NH_MAX_TAO_HASH_NODES;
    } /* hash */

    NH_TAOHashTable taqHashTable;
    NHReturnCode status; // are we
valid
    bool dirty; // have we changed

};

#endif

```



```

//      File:  NHTAQTable.cpp
//
//      Description:
//
//      Implementation to the NHTAQTable class.
//
//
//      History:
//
//      5/14/97      EFB      Created
//      9/9/97      EFB      Added support for culture
//      3/20/98     EFB      Changed names to NH from SN
//

#include <string.h>
#include <stdio.h>

#include      "NHTAQTable.hpp"
#include      "NH_util.hpp"

NHTAQTable::NHTAQTable(NH_TAQ_TABLE_TYPE type)
{
    status = NH_SUCCESS;

    //      clear out the hash table
    for (int i = 0; i < NH_MAX_TAQ_HASH_NODES; i++)
        taqHashTable[i] = NULL;

    //      make sure we are not supposed to be doing an empty table.
    if (type == NH_PRODUCTION_TAQ_TABLE)    {
        //      parameters are:
        //
        //      TAQ string
        //      taq Type (T, P, S, Q, I),
        //      sepIfConjoined ('Y' or 'N')
        //      Given name action      (D - delete, R -
disregard, X - not applicable)
        //      Surname action      (D - delete, R -
disregard, X - not applicable)
        //      Culture (2 char code)

        //      include the data that was generated via the TAQmanager
tool.
        #include      "taqdata.h"

        /*      This stuff is just left over from testing
            addTAQValue("DR", 'T', 'N', NH_TAQ_ACTION_DELETE,
NH_TAQ_ACTION_DELETE, NH_CULTURE_CODE_GENERIC);
            addTAQValue("MR", 'T', 'N', NH_TAQ_ACTION_DELETE,
NH_TAQ_ACTION_DELETE, NH_CULTURE_CODE_GENERIC);
            addTAQValue("MRS", 'T', 'N', NH_TAQ_ACTION_DELETE,
NH_TAQ_ACTION_DELETE, NH_CULTURE_CODE_GENERIC);
            addTAQValue("JR", 'Q', 'N', NH_TAQ_ACTION_DISREGARD,
NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_GENERIC);
            addTAQValue("SR", 'Q', 'N', NH_TAQ_ACTION_DISREGARD,

```

```

NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_GENERIC);
    addTAQValue("ABDUL", 'T', 'N', NH_TAQ_ACTION_DISREGARD,
NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_ARABIC);
    addTAQValue("HOMEY", 'T', 'N', NH_TAQ_ACTION_DISREGARD,
NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_ANGLO);
    addTAQValue("CHINTAQ", 'T', 'N', NH_TAQ_ACTION_DISREGARD,
NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_CHINESE);
    addTAQValue("HISPTAQ", 'T', 'N', NH_TAQ_ACTION_DISREGARD,
NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_HISPANIC);
    addTAQValue("KORTAQ", 'T', 'N', NH_TAQ_ACTION_DISREGARD,
NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_KOREAN);
    addTAQValue("RUSTAQ", 'T', 'N', NH_TAQ_ACTION_DISREGARD,
NH_TAQ_ACTION_DISREGARD, NH_CULTURE_CODE_RUSSIAN);

    */
}

// mark that the table has not been changed. Usefull for
TAQManager application
dirty = false;
}

// release all the memory used to store the NH_TAQRecords
NH_TAQTable::~NH_TAQTable()
{
    NH_TAQRecord *prevTAQRecord;
    NH_TAQRecord *taqRecord;
    int tableIndex;

    for (tableIndex = 0; tableIndex < NH_MAX_TAQ_HASH_NODES;
tableIndex++)
    {
        taqRecord = taqHashTable[tableIndex];
        while (taqRecord != NULL)
        {
            prevTAQRecord = taqRecord;
            taqRecord = taqRecord->next;
            delete prevTAQRecord;
        }
    }
}

// function to take the values passed in, create a NH_TAQRecord
// structure, and add the new structure to this object's
// taqHashTable.
NH_ReturnCode NH_TAQTable::addTAQValue(char *taqValue, char taqType, char
sepIfConjoined,

char gnTAQAction, char snTAQAction, char *taqCulture)
{
    NH_ReturnCode rc = NH_SUCCESS;
    NH_TAQRecord *newTAQRecord;
    int tableIndex;
    NH_TAQRecord *prevTAQRecord;

    // first, make sure we know the culture code
    if (NH_validate_culture_code(taqCulture)) {
        // find the hash value for the taq
        tableIndex = hash(taqValue);

```

```

        // now see if the taq is already defined for this culture
code      // At the same time, find our insertion point, which will
be either:
        // - the first node in the bucket, if this
bucket is empty
        // - the end of the bucket
prevTAQRecord = taqHashTable[tableIndex];
if (prevTAQRecord != NULL) {
    rc = NH_TAQ_ALREADY_EXISTS; // assume
it exists
    while (strcmp(prevTAQRecord->taqString, taqValue) ||
           (strcmp(prevTAQRecord->
>taqCulture, taqCulture))) {
        if (prevTAQRecord->next == NULL) {
            rc = NH_SUCCESS; // does
not exist, so looks good so far
            break; // end of bucket
chain
        }
        prevTAQRecord = prevTAQRecord->
>next; // move through bucket chain
    }

    // if all is still ok (e.g. no duplicate)
    if (rc == NH_SUCCESS) {
        // now create the new record and set its values
        newTAQRecord = new NH_TAQRecord;
        if (newTAQRecord != NULL) {
            NH_safe_strcpy(newTAQRecord->taqString,
taqValue, NH_MAX_TAQ_LEN);
            newTAQRecord->taqType = taqType;
            newTAQRecord->sepIfConjoined = sepIfConjoined;
            newTAQRecord->gnAction = gnTAQAction;
            newTAQRecord->snAction = snTAQAction;
            NH_safe_strcpy(newTAQRecord->taqCulture,
taqCulture, NH_MAX_CULTURE_CODE_LEN);
            newTAQRecord->next = NULL;

            // now add the new record to the chain of
entries (or the start of the
            // bucket. We have already hashed the
tableIndex value above, and have
            // found the correct insertion point
            if (prevTAQRecord == NULL)
                taqHashTable[tableIndex] = newTAQRecord;
            else
                prevTAQRecord->next = newTAQRecord;
        }
        else {
            rc = NH_TAQ_RECORD_ALLOC_ERROR;
            status = NH_TAQ_RECORD_ALLOC_ERROR;
        }
    }
}
else {
    // flag it as an error, but do not mark the entire table
as bad
    rc = NH_INVALID_CULTURE_CODE;

```

```

    }

    return rc;
}

NH_TAQRecordPtr NHTAQTable::getTAQSegment(char *nameSeg, char
*cultureCode)
{
    int                                tableIndex;
    NH_TAQRecordPtr    tempTAQRecordPtr;
    NH_TAQRecordPtr    returnTAQRecordPtr = NULL;

    //    find the hash value for the (possible) taq
    tableIndex = hash(nameSeg);

    //    go throught the records in the chain at that offset in the
    //    hash table, and try to find the taq we are looking for.
    tempTAQRecordPtr = taqHashTable[tableIndex];
    while (tempTAQRecordPtr != NULL)    {
        if (!strcmp(tempTAQRecordPtr->taqString, nameSeg) &&
            !strcmp(tempTAQRecordPtr->taqCulture,
cultureCode))    {
            returnTAQRecordPtr = tempTAQRecordPtr;
            break;
        }
        else //    move on to next record in the chain
            tempTAQRecordPtr = tempTAQRecordPtr->next;
    }
    return returnTAQRecordPtr;
}

//    specialized version of the above function that looks for the
//    name segment in either of the specified culture codes. It makes
//    sure that if the name is found in the primaryCultureCode, that one
//    gets returned even if we come upon the secondaryCultureCode first.
NH_TAQRecordPtr    NHTAQTable::getTAQSegment(char *nameSeg,
char *primaryCultureCode,
char *secondaryCultureCode)
{
    int                                tableIndex;
    NH_TAQRecordPtr    tempTAQRecordPtr;
    NH_TAQRecordPtr    returnTAQRecordPtr = NULL;

    //    find the hash value for the (possible) taq
    tableIndex = hash(nameSeg);

    //    go throught the records in the chain at that offset in the
    //    hash table, and try to find the taq we are looking for.
    tempTAQRecordPtr = taqHashTable[tableIndex];
    while (tempTAQRecordPtr != NULL)    {
        if (!strcmp(tempTAQRecordPtr->taqString, nameSeg) &&
            !strcmp(tempTAQRecordPtr->taqCulture,
primaryCultureCode))    {
            returnTAQRecordPtr = tempTAQRecordPtr;
            break;
        }
    }
}

```

```

        else // move on to next record in the chain
            tempTAQRecordPtr = tempTAQRecordPtr->next;
    }

    // see if we need to check the secondary
    if (returnTAQRecordPtr == NULL) {
        // go through the records in the chain at that offset in
the
        // hash table, and try to find the taq we are looking
for.
        tempTAQRecordPtr = taqHashTable[tableIndex];
        while (tempTAQRecordPtr != NULL) {
            if (!strcmp(tempTAQRecordPtr->taqString, nameSeg) &&
                !strcmp(tempTAQRecordPtr->taqCulture, secondaryCultureCode)) {
                returnTAQRecordPtr = tempTAQRecordPtr;
                break;
            }
            else // move on to next record in the chain
                tempTAQRecordPtr = tempTAQRecordPtr->next;
        }
    }

    return returnTAQRecordPtr;
}

// try to remove the TAQ value specified. If found, return
// NH_SUCCESS. If not found, return.
// The record is deleted if found.
NHReturnCode NHTAQTable::removeTAQValue(char *taqValue, char
*cultureCode)
{
    NHReturnCode rc = NH_TAQ_NOT_FOUND;
    NH_TAQRecordPtr tempTAQRecordPtr;
    NH_TAQRecordPtr prevTAQRecordPtr = NULL;
    int tableIndex =
hash(taqValue);

    // go through the records in the chain at that offset in the
// hash table, and try to find the taq we are looking for.
    tempTAQRecordPtr = taqHashTable[tableIndex];
    while (tempTAQRecordPtr != NULL) {
        if (!strcmp(tempTAQRecordPtr->taqString, taqValue) &&
            !strcmp(tempTAQRecordPtr->taqCulture,
cultureCode))
            break;
        else {
            // save this as the prev
            prevTAQRecordPtr = tempTAQRecordPtr;
            // move on to next record in the chain
            tempTAQRecordPtr = tempTAQRecordPtr->next;
        }
    }

    // once we are here, tempTAQRecordPtr will be non NULL
    // if we found it.
    if (tempTAQRecordPtr != NULL) {
        if (prevTAQRecordPtr == NULL) {
            // this record was the first in the chain, so we
must alter

```

```

        //      the hash table entry
        taqHashTable[tableIndex] = tempTAQRecordPtr->next;
    }
    else          //      not the first in the chain, so assign the
previous one's next
        prevTAQRecordPtr->next = tempTAQRecordPtr-
>next;          //      to our next
        delete tempTAQRecordPtr;
        rc = NH_SUCCESS;
    }

    return rc;
}

```

```

//      File:  NHQueryNameData.cpp
//
//      Description:
//
//      Implementation to the NHQueryNameData class.
//
//
//      History:
//
//      5/14/97      EFB      Created
//      3/20/98      EFB      Changed names to NH from SN
//

```

```

#include <string.h>
#include <stdio.h>

```

```

#include      "NHQueryNameData.hpp"
#include      "NHVariantTable.hpp"
#include      "NHResultsList.hpp"
#include      "NH_util.hpp"
#include      "NHDigraphBitmapArray.hpp"
#include      "NHNameParms.hpp"

```

```

extern      NHDigraphBitmapArray      globalDigraphBitmapArray;

```

```

#define      NH_INDEX_THRESH      0.5

```

```

NHQueryNameData::NHQueryNameData(NHNameParms *nParms, char *aGn, char
*aSn) :
                                NHNameData(nParms, aGn,
aSn)
{
    resultsList = NULL;
    keysArray = NULL;
    numBitsInGnKeys = NULL;
    numBitsInSnKeys = NULL;
    processVariantValues(nParms->gnVariantTable,
nParms->snVariantTable);
}

```

```

NHQueryNameData::NHQueryNameData(NHNameParms *nParms, char *aGn, char
*aSn, char *aMn) :
                                NHNameData(nParms, aGn,
aSn, aMn)
{
    resultsList = NULL;
    keysArray = NULL;
    numBitsInGnKeys = NULL;
    numBitsInSnKeys = NULL;
    processVariantValues(nParms->gnVariantTable,
nParms->snVariantTable);
}

```

```

NHQueryNameData::NHQueryNameData(NHNameParms *nParms, char *name,
NHNameFormat nameFormat)
:
NHNameData(nParms, name,
nameFormat)
{
    resultsList = NULL;
    keysArray = NULL;
    numBitsInGnKeys = NULL;
    numBitsInSnKeys = NULL;
    processVariantValues(nParms->gnVariantTable,
nParms->snVariantTable);
}

NHQueryNameData::~NHQueryNameData()
{
    if (keysArray != NULL)
        delete [] keysArray;
    if (numBitsInGnKeys != NULL)
        delete [] numBitsInGnKeys;
    if (numBitsInSnKeys != NULL)
        delete [] numBitsInSnKeys;
}

// Function to get a pointer to a NHVariant object for each name
// segment. We do this here, in the query
// name, so that lookups only have to be done once for the query
// name.
// Note also that we check first to make sure that we are supposed to
// be
// using variants (we do this per name field).
void NHQueryNameData::processVariantValues(NHVariantTable
*gnVariantTable,
NHVariantTab
le *snVariantTable)
{
    int i;

    if (nameParms->getUseGnVariants()) {
        for (i = 0; i < numGnSegments; i++)
            gnSegmentVariants[i] = gnVariantTable-
>getVariantObjectForName(gnSegments[i].segString);
    }
    if (nameParms->getUseSnVariants()) {
        for (i = 0; i < numSnSegments; i++)
            snSegmentVariants[i] = snVariantTable-
>getVariantObjectForName(snSegments[i].segString);
    }
}

// function to allocate space for, and generate, the keys for
// this query name. The caller calls this explicitly with the
// desired key widths for the GN and SN. We use these
// values in conjunction with the numGnSegments and numSnSegments

```



```

// to calculate how big to make the array that will hold the keys.
void NHQueryNameData::prepareKeys(NHKeyWidth gnKeyWidth,
                                   NHKeyWidth snKeyWidth)
{
    int keyArraySize;
    unsigned char largerNumberOfSegments;
    int fullKeyLen;

    // first allocate the keys
    if (numSnSegments > numGnSegments)
        largerNumberOfSegments = numSnSegments;
    else
        largerNumberOfSegments = numGnSegments;
    if (gnKeyWidth == NH_KEY_WIDTH_32) {
        if (snKeyWidth == NH_KEY_WIDTH_32)
            fullKeyLen = 64;
        else
            fullKeyLen = 96;
    }
    else {
        if (snKeyWidth == NH_KEY_WIDTH_32)
            fullKeyLen = 96;
        else
            fullKeyLen = 128;
    }
    keyArraySize = largerNumberOfSegments * fullKeyLen;
    keysArray = new unsigned int[keyArraySize];

    // save the key lengths
    queryGnKeyWidth = gnKeyWidth;
    querySnKeyWidth = snKeyWidth;

    // now generate the keys for the query
    numBitmapKeys = genIndexKeys(largerNumberOfSegments, gnKeyWidth,
                                   snKeyWidth, keysArray);

    // now allocate space for the arrays that hold the number of
    // bits turned on for each key in the GN and SN.
    numBitsInGnKeys = new unsigned char[largerNumberOfSegments];
    numBitsInSnKeys = new unsigned char[largerNumberOfSegments];

    unsigned char *keysArrayBytePtr = (unsigned char *)keysArray;
    for (int i = 0; i < numBitmapKeys; i++) {
        if (gnKeyWidth == NH_KEY_WIDTH_32) {
            // the number of bits turned on is the sum of the
            number of bits // in each of the 4 bytes that make up the 32 bit
            value
            numBitsInGnKeys[i] =
            globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePtr++) +
            globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
            r++)) +
            globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
            r++)) +
            globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt

```

```

r++));
    }
    else {
        // the number of bits turned on is the sum of the
number of bits // in each of the 8 bytes that make up the 64 bit
value
        numBitsInGnKeys[i] =
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePtr++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++));
    }

    // now do the surname
    if (snKeyWidth == NH_KEY_WIDTH_32) {
        // the number of bits turned on is the sum of the
number of bits // in each of the 4 bytes that make up the 32 bit
value
        numBitsInSnKeys[i] =
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePtr++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++));
    }
    else {
        // the number of bits turned on is the sum of the
number of bits // in each of the 8 bytes that make up the 64 bit
value
        numBitsInSnKeys[i] =
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePtr++)) +
globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +

```

```

        globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(keysArrayBytePt
r++));
    }
}

```

```

#define NH_EITHER_NH_OR_GN 1
#define NH_BOTH_NH_AND_GN 2

```

```

// function to compare the key(s) for this query name against
// a supplied key from an eval name. Before this function is
// called, the caller must have called the
// perpareKeys() method, which sets the gnKeyLength and
// snKeyLength variables, and generates the keys for this
// query name.
// The comparison is performed by looking at the givenname name
// and surname portions of the key separately. For each of these
// subkeys, we see how many bits match, a calculate the quotient of
// matching bits / bits that could have matched. This score is
// compared to ???. If the score for either the GN or SN comparison
// is favorable, the function returns true to indicate that the
// evaluation name associated with the supplied key is a possible
// match, and should be retrieved for further consideration.
// Since this object (the query) could generate multiple keys,
// we may have to perform several comparisons.
bool NHQueryNameData::compareKey(unsigned int *evalBitMapKey, unsigned
char numEvalKeys)
{
    bool rc = false;
    unsigned int *evalKeyPtr;
    unsigned int *queryKeyPtr;
    unsigned int *masterQueryKeyPtr = keysArray;
    unsigned int maskedVal;
    unsigned char numBitsThatMatched;
    unsigned char *bytePtr;
    bool passedGn = false;
    bool passedSn = false;
    int indexMode =
NH_BOTH_NH_AND_GN;

    // for each of the query's keys, do both a SN and GN comparison
    // out nested loop compares the first GN and SN query key to
    // all the eval keys (inner loop), and then moves on to the

```

```

next
    //      query key (outter loop).
    for (int i = 0; (i < numBitmapKeys) && (rc == false); i++) {
        evalKeyPtr = evalBitMapKey;           //      start the
eval ptr at the beggining
        for (int j = 0; j < (int)numEvalKeys; j++) {

            //      place the queryKeyPtr back to the beggining of
the
            //      current query key. This value gets advanced
after we have
            //      compared the current query key to all eval keys
            queryKeyPtr = masterQueryKeyPtr;

            //      first, check the given name
            if (queryGnKeyWidth == NH_KEY_WIDTH_32) {
                //      just compare a 32 bit key for the gn
                maskedVal = *evalKeyPtr & *queryKeyPtr;
                bytePtr = (unsigned char *)&maskedVal;
                numBitsThatMatched =
globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++)) +

                globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +

                globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +

                globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
));

                if ((double)numBitsThatMatched /
(double)numBitsInGnKeys[i] > NH_INDEX_THRESH) {
                    if (indexMode ==
NH_EITHER_NH_OR_GN) {
                        rc = true;
                        break;
                    }
                    else {
//      looking for both, is SN already set?
                        if
(passedSn) {           //      yes, so we matched both
                            rc = true;
                            break;
                        }
                        else
//      no, just set the gn flag
                            passedGn = true;
                    }
                }
                evalKeyPtr++;           //      advance pointers
                queryKeyPtr++;
            }
            else {
                //      just compare a 64 bit key for the gn
                maskedVal = *evalKeyPtr & *queryKeyPtr;
                bytePtr = (unsigned char *)&maskedVal;
                numBitsThatMatched =
globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++)) +

                globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++

```

```

)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
));

    evalKeyPtr++; // advance pointers to get
to second 32 bits in this 64 bit key
    queryKeyPtr++;
    maskedVal = *evalKeyPtr & *queryKeyPtr;
    bytePtr = (unsigned char *)&maskedVal;
    numBitsThatMatched +=
globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
));

    if ((double)numBitsThatMatched /
(double)numBitsInGnKeys[i] > NH_INDEX_THRESH) {
        if (indexMode ==
NH_EITHER_NH_OR_GN) {
            rc = true;
            break;
        }
        else {
// looking for both, is SN already set?
            if
(passedSn) { // yes, so we matched both
                rc = true;
                break;
            }
            else
// no, just set the gn flag
                passedGn = true;
        }
    }
    evalKeyPtr++; // advance pointers
    queryKeyPtr++;
}

// now, check the surname
if (querySnKeyWidth == NH_KEY_WIDTH_32) {
    // just compare a 32 bit key for the sn
    maskedVal = *evalKeyPtr & *queryKeyPtr;
    bytePtr = (unsigned char *)&maskedVal;
    numBitsThatMatched =
globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
    globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
));

```

```

        globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
));

        if ((double)numBitsThatMatched /
(double)numBitsInSnKeys[i] > NH_INDEX_THRESH) {
            if (indexMode ==
NH_EITHER_NH_OR_GN) {
                rc = true;
                break;
            }
            else {
//    looking for both, is GN already set?
                if
(passedGn) { //    yes, so we matched both
                    rc = true;
                    break;
                }
                else
//    no, just set the sn flag
                    passedSn = true;
            }
        }
        evalKeyPtr++; //    advance pointers
        queryKeyPtr++;
    }
    else {
        //    just compare a 64 bit key for the sn
        maskedVal = *evalKeyPtr & *queryKeyPtr;
        bytePtr = (unsigned char *)&maskedVal;
        numBitsThatMatched =
globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
));

        evalKeyPtr++; //    advance pointers to get
to second 32 bits in this 64 bit key
        queryKeyPtr++;
        maskedVal = *evalKeyPtr & *queryKeyPtr;
        bytePtr = (unsigned char *)&maskedVal;
        numBitsThatMatched +=
globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
)) +
        globalDigraphBitmapArray.getNumBitsForByte(*(bytePtr++
));

        if ((double)numBitsThatMatched /
(double)numBitsInSnKeys[i] > NH_INDEX_THRESH) {

```

```

        if (indexMode ==
NH_EITHER_NH_OR_GN) {
            rc = true;
            break;
        }
        else {
//    looking for both, is GN already set?
            if
(passedGn) { //    yes, so we matched both
                rc = true;
                break;
            }
            else
//    no, just set the sn flag
                passedSn = true;
        }
        evalKeyPtr++; //    advance pointers
        queryKeyPtr++;
    }

//    place the master query pointer (for the outer query
loop) at the next
//    query key. We will be advancing the pointer somewhere
between 1 and 4
//    positions (each position is 4 bytes).
    if (queryGnKeyWidth == NH_KEY_WIDTH_32)
        masterQueryKeyPtr++;
    else
        masterQueryKeyPtr += 2;
    if (querySnKeyWidth == NH_KEY_WIDTH_32)
        masterQueryKeyPtr++;
    else
        masterQueryKeyPtr += 2;
}

return rc;
}

```

```

// implementation of SNParmsType helper functions

#include <stdlib.h>
#include <string.h>

#include "NHParmsType.h"

int NH_getParmsTypeIndex(NHParmsType aParmsType);

bool NH_validate_parms_type(NHParmsType aParmsType)
{
    return NH_getParmsTypeIndex(aParmsType) != -1;
}

char * NH_get_culture_string_for_parm_type(NHParmsType aParmsType)
{
    int index = NH_getParmsTypeIndex(aParmsType);
    char *rc;

    if (index != -1)
        rc = NH_culture_strings[index];
    else
        rc = NULL;

    return rc;
}

bool get_culture_code_for_parms_type(NHParmsType aParmsType,
NHCultureCode cultureCode)
{
    bool rc;
    int index = NH_getParmsTypeIndex(aParmsType);

    if (index != -1) {
        strncpy(cultureCode, NH_culture_codes[index],
NH_MAX_CULTURE_CODE_LEN);
        cultureCode[NH_MAX_CULTURE_CODE_LEN] = '\0';
        rc = true;
    }
    else
        rc = false;

    return rc;
}

// function to get the ordinal position of the
// parms type. We use this to then index into the
// NH_culture_codes and NH_culture_strings arrays.
// For this to work, we must make sure that the relative
// order of these enums and arrays stays constant.
int NH_getParmsTypeIndex(NHParmsType aParmsType)
{
    int rc = aParmsType;

```



```
if ((rc < 0) || (rc >= NH_NUM_PARMS_TYPES))  
    rc = -1;  
return rc;
```

```
}
```

```

// File: NHResultsList.cpp
//
// Description:
//
//      Implementation to the NHResultsList class.
//
//
// History:
//
//      6/10/97      EFB      Created
//      3/20/98      EFB      Changed names to NH from SN
//

//      how big should the results list array start out as if
//      they want one that is expandable
#define      NH_DEFAULT_RESULTS_ARRAY_SIZE 2

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#include      "NHResultsList.hpp"
#include      "NHEvalNameData.hpp"
#include      "NH_util.hpp"

static int NH_result_sort_function( const void *arg1, const void *arg2
);

NHResultsList::NHResultsList(int maxHits)
{
    hitArray = NULL;
    isSorted = true;
    numHitsInArray = 0;
    status = NH_SUCCESS;

    if (maxHits > 0) {
        hitArraySize = 2 * maxHits;
        maxHitsToReturn = maxHits;
        hitArray = (NHEvalNameData **)malloc(hitArraySize *
sizeof(NHEvalNameData *));
        if (hitArray == NULL)
            status = NH_RESULTS_LIST_ALLOCATION_ERROR;
    }
    else {
        if (maxHits == NH_RESULTS_LIST_SIZE_EXPANDABLE) {
            //      they want an expandable list
            hitArraySize = NH_DEFAULT_RESULTS_ARRAY_SIZE;
            maxHitsToReturn = NH_RESULTS_LIST_SIZE_EXPANDABLE;
            hitArray = (NHEvalNameData **)malloc(hitArraySize *
sizeof(NHEvalNameData *));
            if (hitArray == NULL)
                status = NH_RESULTS_LIST_ALLOCATION_ERROR;
        }
        else {
            status = NH_INVALID_RESULTS_LIST_SIZE;
            hitArraySize = 0;
        }
    }
}

```

```

        maxHitsToReturn = 0;
    }
}

NHResultsList::~NHResultsList()
{
    for (int i = 0; i < numHitsInArray; i++) {
        delete hitArray[i];
    }
    free(hitArray);
}

NHEvalNameData * NHResultsList::getHitAt(int anIndex)
{
    NHEvalNameData *returnHit;

    // first make sure the list is sorted
    if (isSorted == false) {
        sortHits();
    }

    // now make sure the requested index is valid
    if ((anIndex >= 0) && (anIndex < numHitsInArray))
        returnHit = hitArray[anIndex];
    else
        returnHit = NULL;

    return returnHit;
}

// add a hit to the results list. We make a copy of the hit using
// the default constructor. This should work ok, since do not
// do any dynamic allocation in the class or its subclass. If this
// ever changes, we will need to create a copy constructor for
// the NHEvalNameData and NHNameData classes.
NHReturnCode NHResultsList::addHit(NHEvalNameData *aHit)
{
    NHEvalNameData **newHitArray;
    NHReturnCode rc = NH_SUCCESS;
    NHEvalNameData *hitCopy = new NHEvalNameData(*aHit);

    // *hitCopy = *aHit;
    // if we are supposed to expand
    if (maxHitsToReturn == NH_RESULTS_LIST_SIZE_EXPANDABLE) {
        if (numHitsInArray + 1 == hitArraySize) {
            // we are full, so we must reallocate
            newHitArray = (NHEvalNameData **)realloc(hitArray,
hitArraySize * 2 * sizeof(NHEvalNameData *));
            if (newHitArray != NULL) {
                hitArray = newHitArray;
                hitArraySize *= 2;
            }
        }
        else
            rc = NH_RESULTS_LIST_INSERT_ALLOC_FAILURE;
    }
}

```

```

        if (rc == NH_SUCCESS) {
            hitArray[numHitsInArray] = hitCopy;
            isSorted = false;
            numHitsInArray++;
        }
    }
    else {
        if (hitArray != NULL) {
            hitArray[numHitsInArray] = hitCopy;
            numHitsInArray++;
            isSorted = false;
            // first, make sure our list is not full yet
            if (numHitsInArray >= hitArraySize)
                sortHits();
        }
        else
            rc = NH_RESULTS_ARRAY_NULL_ERROR;
    }
    return rc;
}

// sort the hits, and make sure there are no more than
// maxHitsToReturn items in the array. Any excess items
// should be deleted, and the numHitsInArray variable
// set to be equal to maxHitsToReturn
void NHResultsList::sortHits()
{
    // first, make sure we have something to sort
    if (numHitsInArray > 1) {
        // sort the hits
        qsort(hitArray, numHitsInArray, sizeof(NHEvalNameData *),
              NH_result_sort_function);
        // now, if we have more hits than they wanted, chop some
        off
        // but only chop if we are not expandable
        if (maxHitsToReturn !=
NH_RESULTS_LIST_SIZE_EXPANDABLE) {
            if (numHitsInArray > maxHitsToReturn) {
                for (int i = maxHitsToReturn; i <
numHitsInArray; i++)
                    delete hitArray[i];
                // reflect the new number of hits in the
array
                numHitsInArray = maxHitsToReturn;
            }
        }
        isSorted = true;
    }
}

// return the number of hits. We need to make sure that if
// we are using a fixed size, we do not return a value greater
// than the number they requested.
int NHResultsList::getNumHits(void)
{
    if (maxHitsToReturn == NH_RESULTS_LIST_SIZE_EXPANDABLE)
        return numHitsInArray;
    else
        return numHitsInArray < maxHitsToReturn ?

```

```

        numHitsInArray : maxHitsToReturn;
    }

//    compare function for the results list. Here, we cast the
//    arguments to pointers to NHEvalNameData objects, and compare
//    their scores to see who is larger.
int NH_result_sort_function( const void *arg1, const void *arg2 )
{
    NHEvalNameData *item1 = * (NHEvalNameData **) arg1;
    NHEvalNameData *item2 = * (NHEvalNameData **) arg2;

    return item1->compareScore(item2);
}

```

```

// File: NHNameParms.cpp
//
// Description:
//
// Implementation to the NHNameParms class.
//
//
// History:
//
// 2/27/98 EFB Created to separate pre-processing
parameters from comparison
//
parameters.
// 3/20/98 EFB Changed names to NH from SN
//

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include "NHNameParms.hpp"
#include "NHVariantTable.hpp"
#include "NHTAQTable.hpp"

#include "NH_variant_taq_globals.h"
#include "NHParmsType.h"

NHNameParms::NHNameParms(NHParmsType aParmsType,
bool gnVariants, bool snVariants,
bool gnTaq, bool snTaq,
bool gnUnknowns, bool snUnknowns,
const char *segBreakCharacters,
const char *noiseCharacters)
{
    // assume success
    status = NH_SUCCESS;

    // set these to NULL until we get them
    gnVariantTable = NULL;
    snVariantTable = NULL;
    taqTable = NULL;
    segmentBreakChars = NULL;
    noiseChars = NULL;

    parmsType = aParmsType;

    // set up the culture codes with the one they specified as the
    // primary, and generic as the secondary.
    // This also makes sure the specified culture is valid
    if ((get_culture_code_for_parms_type(aParmsType,
primaryCultureCode) == false) ||
        (get_culture_code_for_parms_type(NH_PARMS_GENERIC,

```

```

secondaryCultureCode) == false))
    status = NH_INVALID_PARMS_TYPE;
else {
    // copy the callers specifications
    useGnVariants = gnVariants;
    useSnVariants = snVariants;
    checkGnUnknowns = gnUnknowns;
    checkSnUnknowns = snUnknowns;
    separateGnTaq = gnTaq;
    separateSnTaq = snTaq;

    // create an artificial loop to cycle through the rest of
the
    // items that need to be created. If more items need to
be
    // added or removed, make sure the 5 changes below).
    for (int i = 0; (i < 5) && (status == NH_SUCCESS);
i++) {
        switch (i) {
            case 0:
                gnVariantTable =
NH_getVariantTable(NH_GIVENNAME_VARIANTS);
                if (gnVariantTable == NULL)
                    status =
NH_GN_VAR_TABLE_CREATION_ERROR;
                else
                    status = gnVariantTable-
>getStatus();
                break;
            case 1:
                snVariantTable =
NH_getVariantTable(NH_SURNAME_VARIANTS);
                if (snVariantTable == NULL)
                    status =
NH_NH_VAR_TABLE_CREATION_ERROR;
                else
                    status = snVariantTable-
>getStatus();
                break;
            case 2:
                taqTable = NH_getTAQTable();
                if (taqTable == NULL)
                    status =
NH_TAQ_TABLE_CREATION_ERROR;
                else
                    status = taqTable->getStatus();
                break;
            case 3:
                // provide a default if they specified
NULL;
                if (segBreakCharacters == NULL)
                    segmentBreakChars =
strdup(NH_DEFAULT_SEG_DELIM_CHARS);
                else
                    segmentBreakChars =
strdup(segBreakCharacters);
                if (segmentBreakChars == NULL)
                    status =
NH_SEG_BREAK_CHARS_CREATION_ERROR;
                break;
            case 4:

```

```

        // provide a default if they specified
NULL;
        if (noiseCharacters == NULL)
            noiseChars =
strdup(NH_DEFAULT_NOISE_CHARS);
        else
            noiseChars =
strdup(noiseCharacters);
        if (noiseChars == NULL)
            status =
NH_NOISE_CHARS_CREATION_ERROR;
        break;
    }
}

// constructor to read from file stream
NHNameParms::NHNameParms(istream &inStream)
{
    status = NH_SUCCESS; // assume success
    segmentBreakChars = NULL;
    noiseChars = NULL;

    if (inStream.good()) {
        inStream.read((char *)&parmsType, sizeof(NHParmsType));
        inStream.read((char *)&useGnVariants, sizeof(bool));
        inStream.read((char *)&useSnVariants, sizeof(bool));
        inStream.read((char *)&separateGnTags, sizeof(bool));
        inStream.read((char *)&separateSnTags, sizeof(bool));
        inStream.read((char *)&checkGnUnknowns, sizeof(bool));
        inStream.read((char *)&checkSnUnknowns, sizeof(bool));

        // write the culture strings.
        inStream.read((char *)primaryCultureCode,
NH_MAX_CULTURE_CODE_LEN + 1);
        inStream.read((char *)secondaryCultureCode,
NH_MAX_CULTURE_CODE_LEN + 1);

        int stringLen;
        char tempString[200 + 1];
        // read string as the length, followed by the null
terminated
        // string, including the NULL
        inStream.read((char *)&stringLen, sizeof(int));

        // make sure we read a reasonable amount from the file
        if (stringLen <= 200) {
            inStream.read((char *)tempString, stringLen + 1);
            setSegmentBreakChars(tempString);

            // write out string as the length, followed by the
null terminated
            // string, including the NULL
            inStream.read((char *)&stringLen, sizeof(int));
            if (stringLen <= 200) {
                inStream.read((char *)tempString, stringLen +
1);
                setNoiseChars(tempString);
            }
        }
    }
}

```



```

        else
            status = NH_NAME_PARMS_FILE_NOISE_CHARS_ERROR;
    }
    else
        status = NH_NAME_PARMS_FILE_BREAKS_CHARS_ERROR;
}
else
    status = NH_NAME_PARMS_BAD_STREAM_ON_CONSTRUCT;

// as a last check, make sure the culture code is valid
if (status == NH_SUCCESS) {
    if (NH_validate_culture_code(primaryCultureCode) == false)
        status = NH_NAME_PARMS_FILE_BAD_CULTURE_CODE;
    else
        if (NH_validate_culture_code(secondaryCultureCode) ==
false)
            status = NH_NAME_PARMS_FILE_BAD_CULTURE_CODE;
}
}

NHNameParms::~NHNameParms()
{
    if (segmentBreakChars != NULL)
        free(segmentBreakChars);
    if (noiseChars != NULL)
        free(noiseChars);
}

// write out the NHNameParms object to a file so that it can
// be read in at a later time.
NHReturnCode NHNameParms::archiveData(ostream &ostream)
{
    NHReturnCode rc = NH_SUCCESS;

    if (ostream.good()) {
        ostream.write((char *)&parmsType, sizeof(NHParmsType));
        ostream.write((char *)&useGnVariants, sizeof(bool));
        ostream.write((char *)&useSnVariants, sizeof(bool));
        ostream.write((char *)&separateGnTags, sizeof(bool));
        ostream.write((char *)&separateSnTags, sizeof(bool));
        ostream.write((char *)&checkGnUnknowns, sizeof(bool));
        ostream.write((char *)&checkSnUnknowns, sizeof(bool));

        // write the culture strings, plus their NULL terminators.
        ostream.write((char *)primaryCultureCode,
NH_MAX_CULTURE_CODE_LEN + 1);
        ostream.write((char *)secondaryCultureCode,
NH_MAX_CULTURE_CODE_LEN + 1);

        int stringLen;
        // write out string as the length, followed by the null
terminated
        // string, including the NULL
        stringLen = strlen(segmentBreakChars);
        ostream.write((char *)&stringLen, sizeof(int));
        ostream.write((char *)segmentBreakChars, stringLen + 1);

        // write out string as the length, followed by the null
terminated
    }
}

```

```

        //      string, including the NULL
        stringLen = strlen(noiseChars);
        outStream.write((char *)&stringLen, sizeof(int));
        outStream.write((char *)noiseChars, stringLen + 1);
    }
    else
        rc = NH_NAME_PARMS_BAD_STREAM_ON_WRITE;
    status = rc;
    return rc;
}

NHReturnCode      NHNameParms::setSegmentBreakChars(char *segBreakChars)
{
    NHReturnCode    retCode = NH_SUCCESS;

    //      first get rid of the old set of characters
    if (segmentBreakChars != NULL) {
        free(segmentBreakChars);
        segmentBreakChars = NULL;
    }

    //      if they gave us a string to set, go ahead and
    //      make a copy of it
    //      If they gave us NULL, make a
    //      copy of an empty string, so we wont have to worry
    //      about accessing a NULL later on.
    if (segBreakChars == NULL)
        segBreakChars = "";
    segmentBreakChars = strdup(segBreakChars);
    if (segmentBreakChars == NULL)
        retCode = NH_SEG_BREAK_CHARS_CREATION_ERROR;

    return retCode;
}

NHReturnCode      NHNameParms::setNoiseChars(char *string)
{
    NHReturnCode    retCode = NH_SUCCESS;

    //      first get rid of the old set of characters
    if (noiseChars != NULL) {
        free(noiseChars);
        noiseChars = NULL;
    }

    //      if they gave us a string to set, go ahead and
    //      make a copy of it.  If they gave us NULL, make a
    //      copy of an empty string, so we wont have to worry
    //      about accessing a NULL later on.
    if (string == NULL)
        string = "";
    noiseChars = strdup(string);
    if (noiseChars == NULL)
        retCode = NH_NOISE_CHARS_CREATION_ERROR;

    return retCode;
}

```

```

// File: NHNameData.cpp
//
// Description:
//
//      Implementation to the NHNameData class.
//
//
// History:
//
//      5/8/97      EFB      Created
//      3/20/98     EFB      Changed names to NH from SN
//

```

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

```

```

#include "NHNameData.hpp"
#include "NHQAQTable.hpp"
#include "NHVariantTable.hpp"
#include "NH_util.hpp"
#include "NHDigraphBitmapArray.hpp"
#include "NHNameParms.hpp"

```

```

extern      NHDigraphBitmapArray      globalDigraphBitmapArray;

```

```

NHNameData::NHNameData(NHNameParms *nParms, char *aGn, char *aSn)
{
    nameStorage = NULL;
    int  gnLen = strlen(aGn);
    int  snLen = strlen(aSn);
    if (gnLen > NH_MAX_GN_LEN)
        gnLen = NH_MAX_GN_LEN;
    if (snLen > NH_MAX_NH_LEN)
        snLen = NH_MAX_NH_LEN;
    allocateNameStorage(gnLen, snLen);
    // NH_safe_strcpy(gn, aGn, NH_MAX_GN_LEN);
    // NH_safe_strcpy(sn, aSn, NH_MAX_NH_LEN);
    NH_safe_strcpy(gn, aGn, gnLen);
    NH_safe_strcpy(sn, aSn, snLen);

    //      save a pointer to the parameters
    nameParms = nParms;

    //      Do the pre-processing on the new name
    preprocessName(nParms->getNoiseChars(), nParms-
>getSegmentBreakChars());
    processTAQValues(nParms->taqTable);
}

```

```

NHNameData::NHNameData(NHNameParms *nParms, char *aGn, char *aSn, char
*aMn)
{
    int  gnLen = strlen(aGn);

```

```

    int    mnLen = strlen(aMn);
    int    snLen = strlen(aSn);
    int    gnAllocLen = gnLen + mnLen + 1;           // space for
the gn, mn and the implied                               // space inbetween

    if (gnAllocLen > NH_MAX_GN_LEN)
        gnAllocLen = NH_MAX_GN_LEN;
    if (snLen > NH_MAX_NH_LEN)
        snLen = NH_MAX_NH_LEN;

    // allocate internal space for the gn and sn
    allocateNameStorage(gnAllocLen, snLen);

// NH_safe_strcpy(gn, aGn, NH_MAX_GN_LEN);
// NH_safe_strcpy(sn, aSn, NH_MAX_NH_LEN);
NH_safe_strcpy(gn, aGn, gnAllocLen);
NH_safe_strcpy(sn, aSn, snLen);

    // now append the middle name onto the gn, but
    // make sure we do not exceed the max allowed chars
    // which is currently the number of chars allowed for
    // GN.
    // We also append a space to the end of the gn so the
    // middle name is separated from it.
    if (gnLen < NH_MAX_GN_LEN) {                       // make
sure there is atleast some room
        strcat(gn + gnLen, " ");
    // NH_safe_strcpy(gn + gnLen + 1, aMn, NH_MAX_GN_LEN - (gnLen +
1));
    // NH_safe_strcpy(gn + gnLen + 1, aMn, gnAllocLen - (gnLen +
1));
    }

    // Things past here are things that need to be done for all
constructors
    // so we may want to move them into a single function that each
of the
    // constructors can call.

    // save a pointer to the parameters
    nameParms = nParms;

    // Do the pre-processing on the new name
    preprocessName(nParms->getNoiseChars(), nParms-
>getSegmentBreakChars());
    processTAQValues(nParms->taqTable);

}

// create a name from a single string. The caller passes in a
// NHNameFormat telling us the format.
// We break the string up into GN and SN. We currently support
// NH_SURNAME_COMMA_GIVENNAME,
// NH_LAST_SEG_IS_SURNAME,
// NH_NAME_FORMAT_UNKNOWN.
//
// For NH_SURNAME_COMMA_GIVENNAME, we place everything to the left of
// a comma in the surname, everything to the right in the given name,

```

```

// and remove the comma. If there is no comma, the entire string
// goes into the given name.
// for NH_LAST_SEG_IS_SURNAME, the last segment becomes the surname,
// and all other segments go into the given name. The process is
smart
// enough to recognize TAQ values when determining the "last"
// segment, so that trailing TAQ values are not considered the last
// segment. Instead, the last non-TAQ segment is treated as the
// last segment. Since the surname should include prefixes and
// suffixes to the "last" segment, we must walk backwards from
// the "last" segment looking for prefixes (TAQ values that occur
// before the "last" segment, and are associated with the "last"
// segment. Some example below help clarify:
//
// Ed Barker SR --> GN =
Ed SN = Barker SR
// Maria De Lahosa Esquire --> GN = Maria SN = De
Lahosa Esquire
// Maria NIETA De Lahosa Esquire --> GN = Maria
NIETA SN = De Lahosa Esquire
// Maria consuala De Lahosa Esquire --> GN = Maria
consuala SN = De Lahosa Esquire
//
// Example1 is simple - SR is a qualifier (suffix), so It gets
placed
// as part of the surname, along with barker, which is the last
// real segment.
//
// Example2 adds the idea of a prefix (De), which gets
associated
// with the last real segment "Lahosa". There is also a
qualifier
// "Esquire" that gets associated with Lahosa.
//
// Example3 differs from example 2 in that there is another TAQ
// value "NIETA", but it is a qualifier that appears before the
// "last" real segment, so it does NOT get associated with the
// Surname.
//
// Example 4 is similar to example 3, except the "consuala"
// segment is not a TAQ value, so it gets associated with the
// Given name (since by default we only use one segment
// for the surname.
//
// If a name is just one
// segment, it becomes the surname, and the given name is blank.
// Currently, NH_NAME_FORMAT_UNKNOWN is treated the same as
// NH_LAST_SEG_IS_SURNAME.
NHNameData::NHNameData(NHNameParms *nParms, char *name, NHNameFormat
nameFormat)
{
    // Need to spilt up the name here, so that the
    // last segment goes into the sn field, and all other
    // parts of the name go into the gn field.

    if (nameFormat == NH_SURNAME_COMMA_GIVENNAME) {
        char *firstComma;

        firstComma = strrchr(name, ',');
        if (firstComma != NULL) {
            // found a comma, so copy everything up to there

```

into the

```
//      sn field
int      snLen = firstComma - name;
int      gnLen = strlen(firstComma + 1);

//      make sure the strings are not too big
if (gnLen > NH_MAX_GN_LEN)
    gnLen = NH_MAX_GN_LEN;
if (snLen > NH_MAX_NH_LEN)
    snLen = NH_MAX_NH_LEN;

//      allocate space for names and segments
allocateNameStorage(gnLen, snLen);

//      make sure the comma was not too far into the
string,
//      i.e. make sure we do not copy to many chars into
the sn
NH_safe_strcpy(sn, name, snLen);

//      copy everything past the comma into the give
name
//      NH_safe_strcpy(gn, firstComma + 1, NH_MAX_GN_LEN);
NH_safe_strcpy(gn, firstComma + 1, gnLen);
NH_strip(gn);
}
else {
    //      no comma found, so put everything in the GN
    //      and blank out the sn.
    int      gnLen = strlen(name);

    if (gnLen > NH_MAX_GN_LEN)
        gnLen = NH_MAX_GN_LEN;
    allocateNameStorage(gnLen, 0);
    NH_safe_strcpy(gn, name, gnLen);
    NH_safe_strcpy(gn, name, NH_MAX_NH_LEN);
    *sn = EOS;
}
}
else {
    //      allocate for a worst case
    allocateNameStorage(NH_MAX_GN_LEN, NH_MAX_NH_LEN);
    //      name format must be NH_LAST_SEG_IS_SURNAME or
NH_NAME_FORMAT_UNKNOWN
    char      *lastSpace;

    //      copy the entire string into the given name
    //      and strip it. We must strip it because we do not
    //      want to find spaces that occur at the end of the name
    //      this would keep us from getting the last segment
properly.
    NH_safe_strcpy(gn, name, NH_MAX_GN_LEN);
    NH_strip(gn);

    lastSpace = strrchr(gn, ' ');
    if (lastSpace != NULL) {
        char      tempSegment[NH_MAX_NH_LEN + 1];
        char      *segmentEndPtr = gn + strlen(gn) -
1;
        //      points to end of last segment
        char      *lastRealSegmentStart = gn;
        //      assume
```

we have all TAQ values,

```

                                                                    //    in which
case we place them

                                                                    //    all into the
SN field.
    NH_TAQRecordPtr  tempTAQRecordPtr; //    pointer to
structure for a TAQ record
    char                                *primaryCultureCod
e = nParms->primaryCultureCode;
    char                                *secondaryCultureC
ode = nParms->secondaryCultureCode;
    NHTAQTable          *taqTable = nParms->taqTable;

    while (lastSpace != NULL)    {
        //    found a space, so see if this segment is a
TAQ value
        NH_safe_strcpy(tempSegment, lastSpace + 1,
segmentEndPtr - lastSpace);
        strupr(tempSegment);

        //    see if this segment is a TAQ value
        tempTAQRecordPtr = taqTable-
>getTAQSegment(tempSegment,

        primaryCultureCode,

        secondaryCultureCode);
        //    if this value was not a TAQ value, then
this segment
        //    is the real last segment.
        if (tempTAQRecordPtr == NULL) {
            lastRealSegmentStart = lastSpace + 1;
            break;
        }
        //    we can safely look at lastSpace - 1
because the entire
        //    gn was stripped, so the leading character
in the gn
        //    could not be a space. Thus, if we have a
space, we are
        //    not at the begining of the string
        segmentEndPtr = lastSpace -
1;    //    get ptr to end of prev segment
        lastSpace = NH_strchr(gn, segmentEndPtr, ' ');
    }

    //    when we are here, lastRealSegmentStart points to
the start
    //    of the last real segment. However, there may be
TAQ values
    //    preceeding the last real segment that are
prefixes or titles,
    //    in which case they should be associated with the
surname.
    if (lastSpace != NULL) {
        segmentEndPtr = lastSpace -
1;    //    get ptr to end of prev segment
    }
```

```

        lastSpace = NH_strchr(gn, segmentEndPtr, ' ');
        if (lastSpace == NULL)
            lastSpace = gn - 1;
1;          // make sure we check the first segment
            while (1) { // breaks get us out of
this loop
                NH_safe_strcpy(tempSegment, lastSpace + 1,
segmentEndPtr - lastSpace);
               strupr(tempSegment);
                tempTAQRecordPtr = taqTable->getTAQSegment(tempSegment,
                    primaryCultureCode,
                    secondaryCultureCode);
                // if this value was not a TAQ value,
then it should // not be associated with the surname,
so just break.
                if (tempTAQRecordPtr == NULL) {
                    break;
                }
                else {
                    // this segment was a TAQ value.
                    // title, we should associated it
                    // just break, so it gets placed
                    if ((tempTAQRecordPtr->taqType !=
                    'P') && (tempTAQRecordPtr->taqType != 'T'))
                        break;
                    else
                        lastRealSegmentStart =
lastSpace + 1; // include this TAQ as part of sn
                }
                if (lastSpace == (gn - 1))
                    break; // already
checked first segment
                else {
                    segmentEndPtr = lastSpace -
1; // get ptr to end of prev segment
                    lastSpace = NH_strchr(gn,
segmentEndPtr, ' ');
                    if (lastSpace == NULL)
                        lastSpace = gn -
1; // make sure we check the first segment
                }
            }
            NH_safe_strcpy(sn, lastRealSegmentStart,
NH_MAX_NH_LEN);
            *lastRealSegmentStart = EOS; // terminate
the GN
        }
        else { // no space found in name, so put
everything in surname field

```



```

        NH_safe_strcpy(sn, gn, NH_MAX_NH_LEN);
        *gn = EOS;
    }

    // save a pointer to the parameters
    nameParms = nParms;

    // Do the pre-processing on the new name
    preprocessName(nParms->getNoiseChars(), nParms-
>getSegmentBreakChars());
    processTAQValues(nParms->taqTable);
}

// constuct an object from an archived representation in
// a stream.
//
// The archive is in the following order
//
// gnLen
// snLen
// nameStorage
NHNameData::NHNameData(NHNameParms.*nParms, istream &inStream)
{
    short int      gnLen;
    short int      snLen;

    // save a pointer to the parameters
    nameParms = nParms;

    // read the given name len and surname len
    inStream.read((char *)&gnLen, sizeof(gnLen));
    if (inStream.gcount() == sizeof(gnLen)) {
        inStream.read((char *)&snLen, sizeof(snLen));

        if (inStream.gcount() == sizeof(snLen)) {
            // allocate space based on the name lengths
            allocateNameStorage(gnLen, snLen);

            // read the name data into the allocated storage.
            inStream.read(nameStorage, variableNameAllocSize);

            // read in the number of gn segments
            if (inStream)
                inStream.read((char *)&numGnSegments,
sizeof(numGnSegments));

            if (inStream) {
                // read in the gn segments. These have been
stored in a special
                // format, with the offset of the segString
as the first item
                unsigned short int      segOffset;
                for (int i = 0; i < numGnSegments; i++) {
                    // first, read in the offset (into the
gnSegs area) of the segString
                    if (inStream)
                        inStream.read((char *)&segOffset,
sizeof(unsigned short int));

```

```

        if (inStream) {
            // update the pointer
            if (segOffset == (unsigned short
int)-1)
                gnSegments[i].segString = "";
            else
                gnSegments[i].segString =
gnSegString + segOffset;
        }

        // now, read in the number of TAQs
        inStream.read((char
*)&(gnSegments[i].numTAQs), sizeof(unsigned char));

        if (inStream) {
            // now read in the TAQs. These
are stored just like the segment,
            // such that the leading element
is the offset of the segString.
            for (int j = 0; j <
gnSegments[i].numTAQs; j++) {
                // first, the offset
                inStream.read((char
*)&segOffset, sizeof(unsigned short int));
                gnSegments[i].taqList[j].segSt
ring = gnSegString + segOffset;

                // next, the TAQ action
                inStream.read((char
*)&(gnSegments[i].taqList[j].taqAction), sizeof(char));

                // lastly, the TAQ type
                inStream.read((char
*)&(gnSegments[i].taqList[j].taqType), sizeof(char));
            }

            // lastly for the segment, the status
            inStream.read((char
*)&(gnSegments[i].status), sizeof(unsigned char));
        }

        // read in the number of sn segments
        inStream.read((char *)&numSnSegments,
sizeof(numSnSegments));

        if (inStream) {
            // do the same thing for the surname
segments
            for (i = 0; i < numSnSegments; i++) {
                // first, read in the offset
(into the snSegs area) of the segString
                inStream.read((char *)&segOffset,
sizeof(unsigned short));

                if (inStream) {
                    // update the pointer
                    if (segOffset == (unsigned
short int)-1)
                        snSegments[i].segString

```

```

= "";
else
    snSegments[i].segString
= snSegString + segOffset;
}

// now, read in the number of
TAQs
    inStream.read((char
*) &(snSegments[i].numTAQs), sizeof(unsigned char));

    if (inStream) {
        // now read in the TAQs.
        // such that the leading
        element is the offset of the segString.
        for (int j = 0; j <
snSegments[i].numTAQs; j++) {
            // first, the offset
            inStream.read((char
*) &segOffset, sizeof(unsigned short));
            snSegments[i].taqList[j]
            .segString = snSegString + segOffset;

            // next, the TAQ
            action
            inStream.read((char
*) &(snSegments[i].taqList[j].taqAction), sizeof(char));

            // lastly, the TAQ
            type
            inStream.read((char
*) &(snSegments[i].taqList[j].taqType), sizeof(char));
        }
        // lastly for the segment, the
        status
        inStream.read((char
*) &(snSegments[i].status), sizeof(unsigned char));
    }
}
else {
    // there was some sort of problem reading in the
    snLen
    // so set nameStorage to NULL so we don't try to
    free it.
    nameStorage = NULL;
}
else {
    // there was some sort of problem reading in the gnLen
    // so set nameStorage to NULL so we don't try to free it.
    nameStorage = NULL;
}
}

```

```

NHNameData::~NHNameData()
{

```

```

    if (nameStorage != NULL)
        free(nameStorage);
}

bool NHNameData::archiveData(ostream &outStream)
{
    bool rc = true;

    // save the given name len and surname len
    outStream.write((char *)&allocatedGnLen, sizeof(allocatedGnLen));
    outStream.write((char *)&allocatedSnLen, sizeof(allocatedSnLen));

    // save the actual name data
    outStream.write(nameStorage, variableNameAllocSize);

    // write out the number of gn segments
    outStream.write((char *)&numGnSegments, sizeof(numGnSegments));

    // write out however many gn segments we need to
    // for each one, we first write out the offset (into the
    // gnSegs area) of the segString member.
    // Then, we write out the numTAQs,
    // then, the TAQs themselves
    // then, the status:
    unsigned short int segOffset;
    for (int i = 0; i < numGnSegments; i++) {
        // first, the segString offset. Check for a null
        // segment, and code it
        // as -1.
        if (gnSegments[i].segString[0] == EOS)
            segOffset = (unsigned short int)-1;
        else
            segOffset = (unsigned short
int)(gnSegments[i].segString - gnSegString);
        outStream.write((char *)&segOffset, sizeof(unsigned short
int));

        // next, number of TAQs
        outStream.write((char *)&(gnSegments[i].numTAQs),
sizeof(unsigned char));

        // next, the TAQs. We do a similar thing here, where we
        first write out
        // the offset of the taq's segString.
        for (int j = 0; j < gnSegments[i].numTAQs; j++) {
            // first, the segString offset
            segOffset = (unsigned short
int)(gnSegments[i].taqList[j].segString - gnSegString);
            outStream.write((char *)&segOffset, sizeof(unsigned
short int));

            // next, the TAQ action
            outStream.write((char
*)&(gnSegments[i].taqList[j].taqAction), sizeof(char));

            // lastly, the TAQ type
            outStream.write((char
*)&(gnSegments[i].taqList[j].taqType), sizeof(char));
        }
    }
}

```

```

        //      lastly for the segment, the status
        outputStream.write((char *)&(gnSegments[i].status),
sizeof(unsigned char));
    }

    //      write out the number of sn segments
    outputStream.write((char *)&numSnSegments, sizeof(numSnSegments));

    //      do the same thing for the sn segments
    for (i = 0; i < numSnSegments; i++) {
        //      first, the segString offset
        if (snSegments[i].segString[0] == EOS)
            segOffset = (unsigned short int)-1;
        else
            segOffset = (unsigned short
int)(snSegments[i].segString - snSegString);
        outputStream.write((char *)&segOffset, sizeof(unsigned short
int));

        //      next, number of TAQs
        outputStream.write((char *)&(snSegments[i].numTAQs),
sizeof(unsigned char));

        //      next, the TAQs. We do a similar thing here, where we
first write out
        //      the offset of the taq's segString.
        for (int j = 0; j < snSegments[i].numTAQs; j++) {
            //      first, the segString offset
            segOffset = (unsigned short
int)(snSegments[i].taqList[j].segString - snSegString);
            outputStream.write((char *)&segOffset, sizeof(unsigned
short int));

            //      next, the TAQ action
            outputStream.write((char
*)&(snSegments[i].taqList[j].taqAction), sizeof(char));

            //      lastly, the TAQ type
            outputStream.write((char
*)&(snSegments[i].taqList[j].taqType), sizeof(char));
        }

        //      lastly for the segment, the status
        outputStream.write((char *)&(snSegments[i].status),
sizeof(unsigned char));
    }

    return rc;
}

```

```

//      go through the different name fields, and remove noise characters
//      Also, convert any segDelimChars to spaces
//      Also, split the name fields into segments
void NHNameData::preprocessName(char *noiseChars, char *segDelimChars)
{
    char *inChar;
    char *outChar;
    int i;

```

```

    numGnSegments = 0;
    inChar = gn;
    outChar = gnSegString;
    *outChar = EOS;
    gnSegments[0].segString = outChar;
    while ((*inChar != EOS) && (numGnSegments <
NH_MAX_SEGS_BEFORE_TAO)) {
        // if this is a noise character, just move on to the next
one in the name
        if (strchr(noiseChars, *inChar))
            inChar++;
        else {
            if (strchr(segDelimChars, *inChar)) {
                // make sure this is not the next in a series
of white spaces
                if (*(gnSegments[numGnSegments].segString) !=
EOS) {
                    // note that we know the segment.
gnSegments[numGnSegments].status =
NH_NAME_FIELD_STATUS_KNOWN;
                    *outChar = EOS; // terminate
the last segment
                    numGnSegments++; // look at next
segment
                    // make sure we are not past the max
number of segments
                    if (numGnSegments >=
NH_MAX_SEGS_BEFORE_TAO)
                        break;
                    inChar++; //
look at next char in name
                    outChar++; // point
to next available space in the output array
                    gnSegments[numGnSegments].segString =
outChar;
                    *outChar = EOS; // init the new
segment
                }
                else // this is a segDelim char, and
so was the last one.
                    inChar++; // so just
ignore it, and move on
            }
            else {
                // just a regular character, so add it to the
segment we are
                // working on currently
                *outChar = toupper(*inChar);
                outChar++; // write to
next character in segment next time.
                inChar++; // look
at next char in name
            }
        }
    }
    // if we get here, it is because we reached the end of the gn
string.
    // If we were in the middle of building a name segment, we
should
    // terminate the segment and increase the number of segments we

```

```

have
    if ((numGnSegments < NH_MAX_SEGS_BEFORE_TAQ) &&
        (*(gnSegments[numGnSegments].segString) !=
EOS)) {
        gnSegments[numGnSegments].status =
NH_NAME_FIELD_STATUS_KNOWN;
        *outChar = EOS;           // terminate the last segment
        numGnSegments++; // look at next segment
    }

    // now do the surname
    numSnSegments = 0;
    inChar = sn;
    outChar = snSegString;
    *outChar = EOS;
    snSegments[0].segString = outChar;
    while ((*inChar != EOS) && (numSnSegments <
NH_MAX_SEGS_BEFORE_TAQ)) {
        // if this is a noise character, just move on to the next
one in the name
        if (strchr(noiseChars, *inChar))
            inChar++;
        else {
            if (strchr(segDelimChars, *inChar)) {
                // make sure this is not the next in a series
of white spaces
                if (*(snSegments[numSnSegments].segString) !=
EOS) {
                    snSegments[numSnSegments].status =
NH_NAME_FIELD_STATUS_KNOWN;
                    *outChar = EOS;           // terminate
the last segment
                    numSnSegments++; // look at next
segment
                    // make sure we are not past the max
number of segments
                    if (numSnSegments >=
NH_MAX_SEGS_BEFORE_TAQ)
                        break;
                    inChar++; //
look at next char in name
                    outChar++; // point
to next available space in the output array
                    snSegments[numSnSegments].segString =
outChar;
                    *outChar = EOS; // init the new
segment
                }
                else // this is a segDelim char, and
so was the last one.
                    inChar++; // so just
ignore it, and move on
            }
            else {
                // just a regular character, so add it to the
segment we are
                // working on currently
                *outChar = toupper(*inChar);
                outChar++; // write to
next character in segment next time.
                inChar++; // look

```

```

at next char in name
    }
}
// if we get here, it is because we reached the end of the sn
string.
// If we were in the middle of building a name segment, we
should
// terminate the segment and increase the number of segments we
have
if ((numSnSegments < NH_MAX_SEGS_BEFORE_TAO) &&
    (*(snSegments[numSnSegments].segString) !=
EOS)) {
    snSegments[numSnSegments].status =
NH_NAME_FIELD_STATUS_KNOWN;
    *outChar = EOS; // terminate the last segment
    numSnSegments++; // look at next segment
}

// now see if there are any segments at all
// in the fields. If not, we should create a
// single blank segment, and mark its status as
// unknown. If there are segments, we need to check for the
// special values NFN, NLN, NMN, FNU, LNU, MNU. If we find
these,
// blank out the segment, and set the status
// appropriately.
// When a name field has more than one segment, but still
// specifies one of these values, we still blank it out,
// but we keep the segment as a blank segment. Although the
// digraph score for this segment will be largely determined by
// the UNKNOWN or NONE parameter, it still gets treated as a
// segment in that oops and anchor val can be applied, and
// it still gets sent to best score.
// We do not currently look across name fields for these
markers.
// That is, we look for NFN, NMN, FNU, MNU in the given name
field
// and we look for NLN and LNU in the surname field.
// ??? Future versions may look across name fields.

if (numGnSegments == 0) {
    numGnSegments = 1;
    gnSegments[0].segString = "";
    gnSegments[0].status = NH_NAME_FIELD_STATUS_UNKNOWN;
}
else if (nameParams->getCheckGnUnknowns()) {
    for (i = 0; i < numGnSegments; i++) {
        if (!strcmp(gnSegments[i].segString, "NFN")) {
            gnSegments[i].segString[0] = EOS;
            gnSegments[i].status =
NH_NAME_FIELD_STATUS_NON_EXISTANT;
        }
        else if (!strcmp(gnSegments[i].segString,
"FNU")) {
            gnSegments[i].segString[0] = EOS;
            gnSegments[i].status =
NH_NAME_FIELD_STATUS_UNKNOWN;
        }
        else if (!strcmp(gnSegments[i].segString,
"NMN")) {
            gnSegments[i].segString[0] = EOS;
            gnSegments[i].status =

```



```

NH_NAME_FIELD_STATUS_NON_EXISTANT;
    } else if (!strcmp(gnSegments[i].segString,
"MNU")){
        gnSegments[i].segString[0] = EOS;
        gnSegments[i].status =
NH_NAME_FIELD_STATUS_UNKNOWN;
    }
}

// now the sn segs
if (numSnSegments == 0) {
    numSnSegments = 1;
    snSegments[0].segString = "";
    snSegments[0].status = NH_NAME_FIELD_STATUS_UNKNOWN;
}
else if (nameParams->getCheckSnUnknowns()) {
    for (i = 0; i < numSnSegments; i++) {
        if (!strcmp(snSegments[i].segString, "NLN")){
            snSegments[i].segString[0] = EOS;
            snSegments[i].status =
NH_NAME_FIELD_STATUS_NON_EXISTANT;
        } else if (!strcmp(snSegments[i].segString,
"LNU")) {
            snSegments[i].segString[0] = EOS;
            snSegments[i].status =
NH_NAME_FIELD_STATUS_UNKNOWN;
        }
    }
}
}

```

```

// function to go through the segments and for each one, see if
// it is a TAQ value. If so, we associate the TAQ with the previous
// or following segment, depending on its type (i.e. prefix, suffix,
etc).
// When we store the TAQ, we also store the action associated with
// the TAQ (currently DELETE or DISREGARD), since this information
// will be required to determine how to adjust the base segment score
//
// Deciding which segment to associate a TAQ with can get pretty
// hairy, especially when mulitple TAQs can be in a name field
// consecutively. We use the Following rules for single TAQ values:
//
// TAQ Type          Segment to Associate with
//
// Prefix            next segment
// Suffix            previous segment
// Infix             Not supported yet
// Title             next segment
// Qualifier         previous segment
//
// These are the basic rules for figuring out which segment to
associate
// TAQs with:
//
// - Any TAQ segments before the first Name segment are
associated with
// the first name segment

```

```

//
// - Any TAQ segments after the last Name segment are associated
with
// the last Name segment
//
// - For TAQs that are surrounded by Name segments :
//
// - All TAQs between a Name segment (on the left) and a
suffix (qualifier)
// (on the right) are associated with the Name Segment.
//
// - All TAQs not fitting the above are associated with the
Name segment
// they proceed.
//
void NHNameData::processTAQValues(NHTAQTable *taqTable)
{
    // NHTAQAction          taqAction;
    int                    i;
    NH_TAQRecordPtr    tempTAQList[NH_MAX_TAQS_PER_SEGMENT];
    // temp list of TAQs found
    int                    tempTAQSegIndex; //
    temp index for the tempTaqList
    NH_TAQRecordPtr    tempTAQRecordPtr; // pointer to structure for
a TAQ record
    int                    numTempTAQSegs;
    // how many TAQs did we find
    int                    segIndex;
    // which segment are we looking at
    int                    lastPrefixIndex; //
    index of last prefix like segment we got
    int                    lastSuffixIndex; //
    index of last suffix like segment we got
    int                    lastNameIndex;
    // index of last non-TAQ segment we got
    int                    nameSegmentTaqListIndex;
    // where to put taqs in a name segments taq list
    char                    *primaryCultureCode =
nameParams->primaryCultureCode;
    char                    *secondaryCultureCode =
nameParams->secondaryCultureCode;

    // clear out the TAQ counts for each segment.
    // This is important because the TAQ segments are not
initialized
    // if they are not filled in.
    for (i = 0; i < numGnSegments; i++)
        gnSegments[i].numTAQs = 0;

    if (nameParams->getSeparateGnTaq() == true) {
        // init some variables
        segIndex = 0;
        numTempTAQSegs = 0;

        // Start out by looking for TAQs at the start of the name
field,
        // before any name segments.
        // while there are TAQ values at the start of the gn
        // get their associated TAQ record and place that in
        // a temporary list.
        while (segIndex < numGnSegments) {

```

```

        tempTAQRecordPtr = taqTable-
>getTAQSegment(gnSegments[segIndex].segString,

        primaryCultureCode,

        secondaryCultureCode);
        if (tempTAQRecordPtr != NULL) {
            // make sure we are not past our space for
TAQs in the temp list
            // This would happen if a name field started
out with tons of TAQs
            if (segIndex < NH_MAX_TAQS_PER_SEGMENT) {
                tempTAQList[numTempTAQSegs] =
tempTAQRecordPtr;
                numTempTAQSegs++;
            }
            segIndex++;
        }
        else
            break;
    }

    // as long as we found a non-TAQ segment
    if (segIndex < numGnSegments) {
        // fill up the taqList for the first Name Segment
with
        // each of the leading TAQs we found. If we found
no TAQs above,
        // numTempTAQSegs will be 0, so we wont even enter
into the loop.
        // Also, since we restricted the loop above, we are
guaranteed to
        // not exceed our space for TAQs for a single
segment.
        for (i = 0; i < numTempTAQSegs; i++) {
            gnSegments[segIndex].taqList[i].segString =
gnSegments[i].segString;
            gnSegments[segIndex].taqList[i].taqAction =
tempTAQList[i]->gnAction;
            gnSegments[segIndex].taqList[i].taqType =
tempTAQList[i]->taqType;
            gnSegments[segIndex].numTAQs += 1;
        }

        // now move all the segments back starting with
first name segment
        // ousting the leading TAQs. If we found that the
first segment
        // was a name segment, we do not need to move
anything.
        if (segIndex != 0) {
            for (i = segIndex; i < numGnSegments;
i++) {
                gnSegments[i - segIndex] = gnSegments[i];
            }
            // note that we now have less segments, since
we removed some segments
            // that were TAQ values
            numGnSegments -= segIndex;
        }
    }

```

```

//      also, set the segIndex to 0, since we are
now back at the beginning
    segIndex = 0;
}

//      now start looking at the remaining segments
//      along the way, we must keep track of
//      -      the index of the last Name segment
we found (start out as 0, since we backed it up to 0)
//      -      the index of the last "suffix-like"
TAQ we found (starts out as -1, since all TAQs were tacked onto seg
0)
//      -      the index of the last "prefix-like"
TAQ we found (starts out as -1, since all TAQs were tacked onto seg
0)
//
//      If we get a:
//      Name:
//      -      associate everything between the
lastNameIndex + 1 and the
//      lastSuffixIndex with
gnSegment[lastNameIndex];
//      -      associate everything between
the lastPrefixIndex and
//      segIndex - 1 with this name
segment.
//      -      move everything back to oust
the TAQ values from the gnSegment array
//      -      mark the new lastNameIndex
(lastNameIndex = segIndex;)
//      -      adjust numGnSegments for how
many TAQs we ousted
//      "Suffix Like"
//      lastPrefixIndex = -
1 //      previous prefix now considered a suffix
//      lastSuffixIndex = segIndex
//      "Prefix Like"
//      lastPrefixIndex =
segIndex
//      End of Segments
//      -      associate everything between the
lastNameIndex + 1 and segIndex
//      with gnSegment[lastNameIndex];
//      -      adjust numGnSegments for how
many TAQs we had at end
//
//      Note that we do not do any storing of anything
until we either reach the
//      end of the sements, or get a non-taq segment.
//
//      Also, as we read TAQ segments, we store a
pointer to their retrieved
//      structure in a list. We do this because we must
read ahead before
//      we can store a TAQs relevant info (type, action)
as being associated
//      with a segment, and we do not want to have to
look up the TAQ info twice.

```

```

        numTempTAQSegs = 0;
        lastPrefixIndex = -1;
        lastSuffixIndex = -1;
        lastNameIndex = segIndex;
        segIndex++; // look at the next segment
        while (segIndex < numGnSegments) {
            tempTAQRecordPtr = taqTable-
>getTAQSegment(gnSegments[segIndex].segString,

        primaryCultureCode,

        secondaryCultureCode);
        if (tempTAQRecordPtr == NULL) {
            // segment is not a TAQ value

            // do an initial check to make sure we
actually got one or more TAQs.
            // if not, all we really have to do is
just reflect the new value for
            // lastNameIndex.
            if (numTempTAQSegs > 0) {
                // so associate all taqs between
the previous Name segment and
                // the last suffix with the
previous Name Segment. Since lastSuffixIndex
                // may be -1 (if there we not
suffixes), we may not even enter this for loop.

                // this variable is necessary
because the segment at lastNameIndex
                // might already have TAQs stored
in its taqList (due to prefixes).
                // We must keep track of where
the next available place in that list is.
                nameSegmentTaqListIndex =
gnSegments[lastNameIndex].numTAQs;
                tempTAQSegIndex = 0;
                for (i = lastNameIndex + 1; (i <=
lastSuffixIndex) && (nameSegmentTaqListIndex < NH_MAX_TAQS_PER_SEGMENT);
i++) {
                    gnSegments[lastNameIndex].taqL
ist[nameSegmentTaqListIndex].segString = gnSegments[i].segString;
                    gnSegments[lastNameIndex].taqL
ist[nameSegmentTaqListIndex].taqAction = tempTAQList[tempTAQSegIndex]-
>gnAction;
                    gnSegments[lastNameIndex].taqL
ist[nameSegmentTaqListIndex].taqType = tempTAQList[tempTAQSegIndex]-
>taqType;
                    tempTAQSegIndex++;
                    nameSegmentTaqListIndex++;
                    gnSegments[lastNameIndex].numT
AQs += 1;
                }

                // associate everything at or
past the previous prefix(s) with the name
                // segment we just found. Again,
since there may not have been any
                // prefixes, we might not even

```

```

        if (lastPrefixIndex != -1) {
            for (i = lastPrefixIndex; (i <
segIndex) && (tempTAQSegIndex < NH_MAX_TAQS_PER_SEGMENT); i++) {
                gnSegments[segIndex].taq
List[i - lastPrefixIndex].segString = gnSegments[i].segString;
                gnSegments[segIndex].taq
List[i - lastPrefixIndex].taqAction = tempTAQList[tempTAQSegIndex]-
>gnAction;
                gnSegments[segIndex].taq
List[i - lastPrefixIndex].taqType = tempTAQList[tempTAQSegIndex]-
>taqType;
                tempTAQSegIndex++;
                gnSegments[segIndex].num
TAQs += 1;

```

```

numGnSegments; i++)      {
numTempTAQSegs};
                        //for (i = lastNameIndex + 1; i <
                        //    gnSegments[i] = gnSegments[i +
                        //}

```

```
// rid of some TAQs
numTempTAQSegs;
too
0;
the temp segment array

segIndex -=
// move our pointer back
numTempTAQSegs =
// clear out
```

```

    }
    else {
        if ((tempTAQRecordPtr->taqType == 'P') ||
(tempTAQRecordPtr->taqType == 'T')) {
            // got a prefix or a title
            tempTAQList[numTempTAQSegs] =
tempTAQRecordPtr;
            numTempTAQSegs++;
            // only set the prefix index if

```

```

we do not have one on record.
the right most prefix in a string
// otherwise, we will only get
// of consecutive prefixes.
if (lastPrefixIndex == -1)
    lastPrefixIndex = segIndex;
}
else {
    // must be a suffix or qualifier
    tempTAQList[numTempTAQSegs] =
tempTAQRecordPtr;
    numTempTAQSegs++;
    lastPrefixIndex = -
1; // any previous prefixes now considered a suffix
    lastSuffixIndex = segIndex;
}
}
segIndex++; // look at next
segment
}
// now we are at the end of all segments, so make
sure that any
// TAQs that were trailing get associated with the
last name segment.
// do an initial check to make sure we actually got
one or more TAQs.
// if not, all we really have to do is just reflect
the new value for
// lastNameIndex.
if (numTempTAQSegs > 0) {
    // associate all the stored taqs with the
last name segment.
    // in the loop below:
    // i is the index into the gnSegments
list for the TAQ string we are copying
    // tempTAQSegIndex is the index into
the tempTAQList for the saved TAQ info
    // lastNameIndex is the index into the
gnSegments for the name getting
    // the TAQs associated with it.
    // gnSegmentTaqListIndex is the index
into the taqList for the name getting
    // the TAQs associated with it.
    //
    // We must be careful that we do not
overwrite any TAQs already associated with
    // the name (from prefixes). For this
reason, we use separate indexes for the
    // tempTAQList and the gnSegments' taqList.

    nameSegmentTaqListIndex =
gnSegments[lastNameIndex].numTAQs;
    tempTAQSegIndex = 0;
    for (i = lastNameIndex + 1; (i < numGnSegments)
&& (nameSegmentTaqListIndex < NH_MAX_TAQS_PER_SEGMENT); i++) {
        gnSegments[lastNameIndex].taqList[nameSegm
entTaqListIndex].segString = gnSegments[i].segString;
        gnSegments[lastNameIndex].taqList[nameSegm
entTaqListIndex].taqAction = tempTAQList[tempTAQSegIndex]->gnAction;

```

```

        gnSegments[lastNameIndex].taqList[nameSegmentTaqListIndex].taqType = tempTaqList[tempTAQSegIndex]->taqType;
        tempTAQSegIndex++;
        nameSegmentTaqListIndex++;
        gnSegments[lastNameIndex].numTAQs += 1;
    }

    // now we can just chop off all the TAQ
segments by reducing numGnSegments.
    numGnSegments -= numTempTAQSegs;
}

}
else {
    // we did not get any Non-TAQ segments. Move all
the segments to the TAQ
    // list for the first segment, create a single
segment, and set its string
    // value to "".
    gnSegments[0].numTAQs = 0; // set this in case
there were no TAQs (empty string)

    // In that case, we would not have
cleared it out originally
    for (i = 0; i < numTempTAQSegs; i++) {
        gnSegments[0].taqList[i].segString =
gnSegments[i].segString;
        gnSegments[0].taqList[i].taqAction =
tempTaqList[i]->gnAction;
        gnSegments[0].taqList[i].taqType =
tempTaqList[i]->taqType;
        gnSegments[0].numTAQs += 1;
    }
    numGnSegments = 1;
    gnSegments[0].segString = "";
    gnSegments[0].status = NH_NAME_FIELD_STATUS_UNKNOWN;
}

}

// as a last step, we must make sure that the number of
gnSegments is
// now no greater than NH_MAX_SEGS_AFTER_TAQ. We just ignore
any segments
// after the max.
if (numGnSegments > NH_MAX_SEGS_AFTER_TAQ)
    numGnSegments = NH_MAX_SEGS_AFTER_TAQ;

// clear out the TAQ counts for each segment.
// This is important because the TAQ segments are not
initialized
// if they are not filled in.
for (i = 0; i < numSnSegments; i++)
    snSegments[i].numTAQs = 0;

// Now do the SN segments
if (nameParms->getSeparateGnTaq() == true) {
    // init some variables
    segIndex = 0;
    numTempTAQSegs = 0;
}

```



```

        //      Start out by looking for TAQs at the start of the name
field,        //      before any name segments.
        //      while there are TAQ values at the start of the sn
        //      get their associated TAQ record and place that in
        //      a temporary list.
        while (segIndex < numSnSegments) {
            tempTAQRecordPtr = taqTable-
>getTAQSegment(snSegments[segIndex].segString,

            primaryCultureCode,

            secondaryCultureCode);
            if (tempTAQRecordPtr != NULL) {
                //      make sure we are not past our space for
TAQs in the temp list        //      This would happen if a name field started
out with tons of TAQs        if (segIndex < NH_MAX_TAQS_PER_SEGMENT) {
                                tempTAQList[numTempTAQSegs] =
tempTAQRecordPtr;
                                numTempTAQSegs++;
                                }
                                segIndex++;
                            }
                            else
                                break;
                        }

                        //      as long as we found a non-TAQ segment
                        if (segIndex < numSnSegments) {
                            //      fill up the taqList for the first Name Segment
with                            //      each of the leading TAQs we found. If we found
no TAQs above,                //      numTempTAQSegs will be 0, so we wont even enter
into the loop.                //      Also, since we restricted the loop above, we are
guaranteed to                //      not exceed our space for TAQs for a single
segment.                      for (i = 0; i < numTempTAQSegs; i++) {
                                snSegments[segIndex].taqList[i].segString =
snSegments[i].segString;
                                snSegments[segIndex].taqList[i].taqAction =
tempTAQList[i]->snAction;
                                snSegments[segIndex].taqList[i].taqType =
tempTAQList[i]->taqType;
                                snSegments[segIndex].numTAQs += 1;
                                }

                                //      now move all the segments back starting with
first name segment           //      ousting the leading TAQs. If we found that the
first segment                //      was a name segment, we do not need to move
anything.                    if (segIndex != 0) {
                                for (i = segIndex; i < numSnSegments;

```

```

i++) {
    snSegments[i - segIndex] = snSegments[i];
    }
    // note that we now have less segments, since
we removed some segments
    // that were TAQ values
    numSnSegments -= segIndex;

    // also, set the segIndex to 0, since we are
now back at the beginning
    segIndex = 0;
}

// now start looking at the remaining segments
// along the way, we must keep track of
// - the index of the last Name segment
we found (start out as 0, since we backed it up to 0)
// - the index of the last "suffix-like"
TAQ we found (starts out as -1, since all TAQs were tacked onto seg
0)
// - the index of the last "prefix-like"
TAQ we found (starts out as -1, since all TAQs were tacked onto seg
0)
//
// If we get a:
// Name:
// - associate everything between the
lastNameIndex + 1 and the
// lastSuffixIndex with
snSegment[lastNameIndex];
// - associate everything between
the lastPrefixIndex and
// segIndex - 1 with this name
segment.
// - move everything back to oust
the TAQ values from the snSegment array
// - mark the new lastNameIndex
(lastNameIndex = segIndex;)
// - adjust numSnSegments for how
many TAQs we ousted
// "Suffix Like"
// lastPrefixIndex = -
1 // previous prefix now considered a suffix
// lastSuffixIndex = segIndex
// "Prefix Like"
// lastPrefixIndex =
segIndex
// End of Segments
// - associate everything between the
lastNameIndex + 1 and segIndex
// with snSegment[lastNameIndex];
// - adjust numSnSegments for how
many TAQs we had at end
//
// Note that we do not do any storing of anything
until we either reach the
// end of the segments, or get a non-taq segment.
//
// Also, as we read TAQ segments, we store a
pointer to their retrieved
// structure in a list. We do this because we must

```

```

read ahead before
// we can store a TAQs relevant info (type, action)
as being associated
// with a segment, and we do not want to have to
look up the TAQ info twice.

    numTempTAQSegs = 0;
    lastPrefixIndex = -1;
    lastSuffixIndex = -1;
    lastNameIndex = segIndex;
    segIndex++; // look at the next segment
    while (segIndex < numSnSegments) {
        tempTAQRecordPtr = taqTable-
>getTAQSegment(snSegments[segIndex].segString,

        primaryCultureCode,

        secondaryCultureCode);
        if (tempTAQRecordPtr == NULL) {
            // segment is not a TAQ value
            // do an initial check to make sure we
actually got one or more TAQs.
            // if not, all we really have to do is
just reflect the new value for
            // lastNameIndex.
            if (numTempTAQSegs > 0) {
                // so associate all taqs between
the previous Name segment and
                // the last suffix with the
previous Name Segment. Since lastSuffixIndex
                // may be -1 (if there we not
suffixes), we may not even enter this for loop.

                // this variable is necessary
because the segment at lastNameIndex
                // might already have TAQs stored
in its taqList (due to prefixes).
                // We must keep track of where
the next available place in that list is.
                nameSegmentTaqListIndex =
snSegments[lastNameIndex].numTAQs;
                tempTAQSegIndex = 0;
                for (i = lastNameIndex + 1; (i <=
lastSuffixIndex) && (nameSegmentTaqListIndex < NH_MAX_TAQS_PER_SEGMENT);
i++) {
                    snSegments[lastNameIndex].taqL
ist[nameSegmentTaqListIndex].segString = snSegments[i].segString;
                    snSegments[lastNameIndex].taqL
ist[nameSegmentTaqListIndex].taqAction = tempTAQList[tempTAQSegIndex]-
>snAction;
                    snSegments[lastNameIndex].taqL
ist[nameSegmentTaqListIndex].taqType = tempTAQList[tempTAQSegIndex]-
>taqType;
                    tempTAQSegIndex++;
                    nameSegmentTaqListIndex++;
                    snSegments[lastNameIndex].numT
AQs += 1;

```

```

    }

    // associate everything at or
    past the previous prefix(s) with the name
    // segment we just found. Again,
    since there may not have been any
    // prefixes, we might not even
    enter this for loop
        if (lastPrefixIndex != -1) {
            for (i = lastPrefixIndex; (i <
                segIndex) && (tempTAQSegIndex < NH_MAX_TAQS_PER_SEGMENT); i++) {
                snSegments[segIndex].taq
                List[i - lastPrefixIndex].segString = snSegments[i].segString;
                snSegments[segIndex].taq
                List[i - lastPrefixIndex].taqAction = tempTAQList[tempTAQSegIndex]-
                >snAction;
                snSegments[segIndex].taq
                List[i - lastPrefixIndex].taqType = tempTAQList[tempTAQSegIndex]-
                >taqType;
                tempTAQSegIndex++;
                snSegments[segIndex].num
            }
        }

    TAQs += 1;

    // now move all the segments back
    starting with this segment and
    // ending with the last segment.
    We move them back to the first
    // segment after the previous
    Name segment, which is numTempTAQSegs places
    for (i = segIndex; i <
    numSnSegments; i++) {
        snSegments[i - numTempTAQSegs]
        = snSegments[i];
    }

    numSnSegments -=
    numTempTAQSegs; // we not have less segments, since we got

    // rid of some TAQs
    segIndex -=
    numTempTAQSegs; // move our pointer back
    too
    numTempTAQSegs =
    0; // clear out
    the temp segment array

    }
    lastNameIndex =
    segIndex; // mark the new
    lastNameIndex

    }
    else {
        if ((tempTAQRecordPtr->taqType == 'P') ||
            (tempTAQRecordPtr->taqType == 'T')) {
            // got a prefix or a title
            tempTAQList[numTempTAQSegs] =
            tempTAQRecordPtr;
            numTempTAQSegs++;
        }
    }

```

```

// only set the prefix index if
we do not have one on record.
// otherwise, we will only get
the right most prefix in a string
// of consecutive prefixes.
if (lastPrefixIndex == -1)
    lastPrefixIndex = segIndex;
}
else {
    // must be a suffix or qualifier
    tempTAQList[numTempTAQSegs] =
tempTAQRecordPtr;
    numTempTAQSegs++;
    lastPrefixIndex = -
1; // any previous prefixes now considered a suffix
    lastSuffixIndex = segIndex;
}
}
segIndex++; // look at next
segment
}

// now we are at the end of all segments, so make
sure that any
// TAQs that were trailing get associated with the
last name segment.

// do an initial check to make sure we actually got
one or more TAQs.
// if not, all we really have to do is just reflect
the new value for
// lastNameIndex.
if (numTempTAQSegs > 0) {
    // associate all the stored taqs with the
last name segment.
    // in the loop below:
    // i is the index into the snSegments
list for the TAQ string we are copying
    // tempTAQSegIndex is the index into
the tempTAQList for the saved TAQ info
    // lastNameIndex is the index into the
snSegments for the name getting
    // the TAQs associated with it.
    // snSegmentTaqListIndex is the index
into the taqList for the name getting
    // the TAQs associated with it.
    //
    // We must be careful that we do not
overwrite any TAQs already associated with
    // the name (from prefixes). For this
reason, we use separate indexes for the
    // tempTAQList and the snSegments' taqList.

    nameSegmentTaqListIndex =
snSegments[lastNameIndex].numTAQs;
    tempTAQSegIndex = 0;
    for (i = lastNameIndex + 1; (i < numSnSegments)
    && (nameSegmentTaqListIndex < NH_MAX_TAQS_PER_SEGMENT); i++) {
        snSegments[lastNameIndex].taqList[nameSegm
entTaqListIndex].segString = snSegments[i].segString;
        snSegments[lastNameIndex].taqList[nameSegm

```

```

entTaqListIndex].taqAction = tempTAQList[tempTAQSegIndex]->snAction;
snSegments[lastNameIndex].taqList[nameSegmentTaqListIndex].taqType = tempTAQList[tempTAQSegIndex]->taqType;
tempTAQSegIndex++;
nameSegmentTaqListIndex++;
snSegments[lastNameIndex].numTAQs += 1;
}

// now we can just chop off all the TAQ
segments by reducing numSnSegments.
numSnSegments -= numTempTAQSegs;
}

}
else {
// we did not get any Non-TAQ segments. Move all
the segments to the TAQ
// list for the first segment, create a single
segment, and set its string
// value to "".
snSegments[0].numTAQs = 0; // set this in case
there were no TAQs (empty string)

// In that case, we would not have
cleared it out originally
for (i = 0; i < numTempTAQSegs; i++) {
snSegments[0].taqList[i].segString =
snSegments[i].segString;
snSegments[0].taqList[i].taqAction =
tempTAQList[i]->snAction;
snSegments[0].taqList[i].taqType =
tempTAQList[i]->taqType;
snSegments[0].numTAQs += 1;
}
numSnSegments = 1;
snSegments[0].segString = "";
snSegments[0].status = NH_NAME_FIELD_STATUS_UNKNOWN;
}

// as a last step, we must make sure that the number of
gnSegments is
// now no greater than NH_MAX_SEGS_AFTER_TAQ. We just ignore
any segments
// after the max.
if (numSnSegments > NH_MAX_SEGS_AFTER_TAQ)
numSnSegments = NH_MAX_SEGS_AFTER_TAQ;
}

// function to generate index keys for this name.
// Each key includes a portion for the GN and a portion
// for the SN.
// We currently support two key lengths, 32 bits or 64 bits.
// The GN length does not have to be the same as the SN length,
// but GN keys generated must be the same length (similarly for
// SN). Thus the full key length could be:
//
// 64: Both GN and SN are 32 bits

```

```

//          96:          Gn is 64, but SN is 32
//          96:          Gn is 32, but SN is 64
//          128: Both GN and SN are 64 bits
//
// Keys are generated by name stem segment. The first key
// consists of a key for the first GN segment, and a key
// for the first SN segment. The second key
// consists of a key for the second GN segment, and a key
// for the second SN segment. When there are a differing number
// of GN and SN segments, the final segment of the name
// field with the fewer number of segments is repeated.
// Thus, the number of keys generated is given by the formula:
//          max(numGnSegs, numSnSegs)
//
// We do things this way so that a name has the same number of keys
// for both GN and SN, and in fact we can view the two keys as one
// contiguous key that can be passed to comparison functions as a
// single value.
//
// Note that we are talking about stem segments (TAQ segments have
// been removed).
//
// maxKeys specifies how many keys the caller can fit into
// keyBuff. It is up to the caller to make sure that they have
// allocated
// enough space in the keyBuff to hold maxKeys.
unsigned char NHNameData::genIndexKeys(int maxKeys, NHKeyWidth
gnKeyWidth,
                                     NHKeyWidth snKeyWidth, void *keyBuff)
{
    int numKeysGenerated = 0;
    int gnSegIndex = 0;
    int snSegIndex = 0;
    unsigned int *keyPtr = (unsigned int *)keyBuff;

    while (numKeysGenerated < maxKeys) {
        if ((gnSegIndex >= numGnSegments) && (snSegIndex >=
numSnSegments))
            break;
        else {
            numKeysGenerated++;
            // make sure that if one segment is now at the end,
            // we stay on the last segment
            if (gnSegIndex == numGnSegments)
                gnSegIndex--;
            if (snSegIndex == numSnSegments)
                snSegIndex--;

            if (gnKeyWidth == NH_KEY_WIDTH_32) {
                // gn key length is 32
                *keyPtr =
globalDigraphBitmapArray.get32BitKeyForToken(gnSegments[gnSegIndex].segS
tring);

                keyPtr++; // move the pointer by 4
bytes
            }
            else {
                // gn key length is 64

```

```

                                globalDigraphBitmapArray.get64BitKeyForToken(gnS
gments[gnSegIndex].segString,

```

```

        (bit_64_t *)keyPtr);
        keyPtr += 2;           // move the pointer
by 8 bytes

```

```

        if (snKeyWidth == NH_KEY_WIDTH_32) {
            //      gn key length is 32
            *keyPtr =
globalDigraphBitmapArray.get32BitKeyForToken(snSegments[snSegIndex].segS
tring);

```

```

    keyPtr++; // move the pointer by 4
    bytes
    }
    else {
        // gn key length is 64
        globalDigraphBitmapArray.get64BitKeyForToken(snS
egments[snSegIndex].segString,

```

```

        (bit_64_t *)keyPtr);
        keyPtr += 2;           // move the pointer
by 8 bytes
    }
}

```

```
//      advance the segment indexes
snSegIndex++;
gnSegIndex++;
    }
}
return numKeysGenerated;
}
```



```

// File: NHEvalNameData.cpp
//
// Description:
//
//      Implementation to the NHEvalNameData class.
//
// History:
//
//      5/14/97      EFB      Created
//      9/1/97      EFB      Lots of changes to support
retaining segment scores in
//                                best mode so
that sorting can be more detailed and accurate
//      10/31/97    EFB      Made several member functions
protected, and made performComp()
//                                a friend of
NHQueryNameData. Also changed performComp to
//                                NOT delete
objects that are not passed on to the resultslist,
//                                to
acomodate the new method of deleting NHEvalNameData objects.
//      11/03/97    EFB      Added a new function,
calcNameScore() and made it virtual.
//                                removed
virtual from performComp. The perform comp method
//                                was too
complicated to be subclassed. We really only want
//                                callers to
be able to affect the name score and the determination
//                                of
HIT/NO_HIT. These are now the only virtual functions. Both
//                                are now
inline in the header file so the caller knows exactly
//                                what is
happening in these functions if they decide to subclass
//                                and
override. OOPS, I forgot compareScore(), which is also
//                                virtual - we
want them to be able to change how hits are sorted.
//
//      3/02/98      EFB      Made lots of changes necessary
when I moved a bunch of
//                                parameters
(the ones associated with parsing the name)
//                                from the
NHCompParms class into a new class called NHNameParms.
//                                and renamed
the NHCompParms class to NHCompParms.
//      3/20/98      EFB      Changed names to NH from SN

```

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#include "NHEvalNameData.hpp"
#include "NHQueryNameData.hpp"
#include "NH_util.hpp"
#include "NH_queens_arrays.hpp"

```

```

#include "NHVariantTable.hpp"
#include "NHResultsList.hpp"
#include "NHQAQTable.hpp"
#include "NHNameParms.hpp"

// private, non-member function prototype
static double NH_digraph_score(char *qSeg, int qSegLen,
char *evalSeg, int evalSegLen,
bool useLeftDigraphBias);

static double NH_best_score(int numQSegs, int numEvalSegs,
NHSegScoreMode scoreMode, double
scores[NH_MAX_SEGS_AFTER_TAO][NH_MAX_SEGS_AFTER_TAO]);

void NH_best_score_for_highest_mode(int xDim, int yDim, double
highestScore, double
*bestSegScores, double
scores[NH_MAX_SEGS_AFTER_TAO][NH_MAX_SEGS_AFTER_TAO]);

static double NH_calc_score(SegList qSegs, int numQSegs, SegLis
t evalSegs, int numEvalSegs, SegLis
tVariants querySegmentVariants, char
*primaryCulture, char
*secondaryCulture, NHComp
Parms *compParms, NHName
Parms *nameParms, NHName
Fields nameField, char
*origQNameField, char
*origEvalNameField, int
*numSegsScored, double
*bestSegScores);

static void NH_apply_TAQs_to_score(double *diScore, Segment *qSeg,
Segment *evalSeg,
double absDeltaTAQFactor,
double absDistTAQFactor,
double deltaTAQFactor,
double distTAQFactor);

static bool NH_check_compressed_name(char *qSegString, char

```

```
*evalSegString,
```

```
char *compressCharsPart1,
```

```
char *compressCharsPart2);
```

```
NHEvalNameData::NHEvalNameData(NHNameParms *nParms, char *aGn, char  
*aSn) :
```

```
NHNameData(nParms, aGn,
```

```
aSn)
```

```
{
```

```
    resetScores();
```

```
}
```

```
NHEvalNameData::NHEvalNameData(NHNameParms *nParms, char *aGn, char  
*aSn, char *aMn) :
```

```
NHNameData(nParms, aGn,
```

```
aSn, aMn)
```

```
{
```

```
    resetScores();
```

```
}
```

```
NHEvalNameData::NHEvalNameData(NHNameParms *nParms, char *name,  
NHNameFormat nameFormat) :
```

```
NHNameData(nParms, name,
```

```
nameFormat)
```

```
{
```

```
    resetScores();
```

```
}
```

```
//    constuct an object from an archived representation in  
//    a stream.
```

```
//
```

```
//    The archive is in the following order
```

```
//
```

```
//    gnLen
```

```
//    snLen
```

```
//    nameStorage
```

```
NHEvalNameData::NHEvalNameData(NHNameParms *nParms, istream &inStream) :  
NHNameData(nParms,
```

```
inStream)
```

```
{
```

```
    //    read the gn, sn and name scores
```

```
    if (inStream)
```

```
        inStream.read((char *)&gnScore, sizeof(gnScore));
```

```
    if (inStream)
```

```
        inStream.read((char *)&snScore, sizeof(snScore));
```

```
    if (inStream)
```

```
        inStream.read((char *)&nameScore, sizeof(nameScore));
```

```
    //    seg differentials
```

```
    if (inStream)
```

```
        inStream.read((char *)&gnSegDifferential,  
sizeof(gnSegDifferential));
```

```
    if (inStream)
```

```
        inStream.read((char *)&snSegDifferential,  
sizeof(snSegDifferential));
```

```

        //      read the number of gn segs scored, and however many scores
we need      inStream.read((char *)&numGnSegsScored,
sizeof(numGnSegsScored));
        if (inStream)
            inStream.read((char *)&numGnSegsScored,
sizeof(numGnSegsScored));
        if (inStream) {
            if (numGnSegsScored > 0) {
                inStream.read((char *)gnSegScores, numGnSegsScored *
sizeof(double));
            }
        }

        //      read the number of sn segs scored, and however many scores
we need
        if (inStream)
            inStream.read((char *)&numSnSegsScored,
sizeof(numSnSegsScored));
        if (inStream) {
            if (numSnSegsScored > 0) {
                inStream.read((char *)snSegScores, numSnSegsScored *
sizeof(double));
            }
        }
    }

NHEvalNameData::~NHEvalNameData()
{
}

bool NHEvalNameData::archiveData(ostream &outStream)
{
    bool rc = true;

    rc = NHNameData::archiveData(outStream);
    if (rc) {
        //      read the gn, sn and name scores
        outStream.write((char *)&gnScore, sizeof(gnScore));
        outStream.write((char *)&snScore, sizeof(snScore));
        outStream.write((char *)&nameScore, sizeof(nameScore));

        //      seg differentials
        outStream.write((char *)&gnSegDifferential,
sizeof(gnSegDifferential));
        outStream.write((char *)&snSegDifferential,
sizeof(snSegDifferential));

        //      read the number of gn segs scored, and however many
scores we need      inStream.read((char *)&numGnSegsScored,
sizeof(numGnSegsScored));
        outStream.write((char *)&numGnSegsScored,
sizeof(numGnSegsScored));
        if (numGnSegsScored > 0) {
            outStream.write((char *)gnSegScores, numGnSegsScored *
sizeof(double));
        }
    }
}

```

[illegible]

```

SegsScored,                                     &numSn
                                                snSegS
cores);
}

//    note that this function is a friend of NHQueryNameData, which is
//    why we are able to access private member functions of that class.
NHReturnCode    NHEvalNameData::performComp(NHQueryNameData
*queryName,

                                                NHCompParms
*someCompParms)
{
    NHReturnCode    compResult;
    NHResultsList   *resultList;

    //    save the compParms so that they can be easily referenced
    //    throughout the comparison process.
    compParms = someCompParms;

    calcComponentScores(queryName);

    //    call a method to calculate the name score.
    calcNameScore();

    //    store the segments differentials, in case we get a tie
score.
    gnSegDifferential = abs(numGnSegments - queryName-
>getNumGnSegments());
    snSegDifferential = abs(numSnSegments - queryName-
>getNumSnSegments());

    //    Now call the getCompResult() function to get the return
value
    //    (i.e. was it a match?)
    compResult = getCompResult();

    //    now see if we are working with a results list
    resultList = queryName->getResultsList();
    if (resultList != NULL) {
        //    we are using a result list.  If this is a hit, add it
        //    to the result list.
        //    Otherwise, delete it
        if (compResult == NH_MATCH) {
            NHReturnCode    tempInsertResult;

            //    make sure the insert works.  If so, don't mess
with
            //    the compResult, so the comparison will be
returned
            //    as a hit.  If there was an error, delete this
object,
            //    and save the error code so it can be returned.
            tempInsertResult = resultList->addHit(this);
            if (tempInsertResult != NH_SUCCESS) {
                compResult = tempInsertResult;
            }
        }
    }
}

```

```

    }
    return compResult;
}

// used only when the segment mode is set to HIGHEST.
// It compares the segment scores the were retained when
// the name was compared to the query name.
// We are comparing the segment scores for two (pre-scored)
// eval names. The comparison should find which name has
// the "best" set of segment scores, where best is defined
// as "the one with the highest best score". If the best
// score results in a tie, we move on to the second best score,
// and so on until we find a difference, or there are no more
// segments to compare. Each name has variables numGnSegsScored
// and numSnSegsScored, that tell how many segments were scored
// in the name. We do up to N comparisons, where N is the larger
// of the number of segments scored in each name. Where one name
// has less segments scored than the other, a default value of
// NH_DEFAULT_MISSING_SEGMENT_SCORE is assigned. This is so that
// a scored segment has to beat some threshold to be considered
// better than nothing at all.
//
double NHEvalNameData::compareSegmentScores(NHEvalNameData
*scoredName, NHNameFields nameField)
{
    double scoreDiff;
    int maxComparisons;
    double *thisEvalScores;
    double *compEvalScores;
    int numSegsScoredForThisEval;
    int numSegsScoredForCompEval;

    if (nameField == NH_LAST_NAME) {
        thisEvalScores = snSegScores;
        compEvalScores = scoredName->snSegScores;
        numSegsScoredForThisEval = numSnSegsScored;
        numSegsScoredForCompEval = scoredName->numSnSegsScored;
    }
    else {
        thisEvalScores = gnSegScores;
        compEvalScores = scoredName->gnSegScores;
        numSegsScoredForThisEval = numGnSegsScored;
        numSegsScoredForCompEval = scoredName->numGnSegsScored;
    }
    maxComparisons = numSegsScoredForThisEval >
numSegsScoredForCompEval ? numSegsScoredForThisEval :
numSegsScoredForCompEval;

    for (int i = 0; i < maxComparisons; i++) {
        if (i >= numSegsScoredForThisEval)
            thisEvalScores[i] = NH_DEFAULT_MISSING_SEGMENT_SCORE;
        else // we can do an else because only one segment
can be missing, not both
            if (i >= numSegsScoredForCompEval)
                compEvalScores[i] =
NH_DEFAULT_MISSING_SEGMENT_SCORE;

        scoreDiff = compEvalScores[i] - thisEvalScores[i];
        if (scoreDiff != 0)

```

```

break;
}

return scoreDiff;
}

/*****
****/

/* NH_calc_score
   Performs a string comparison on two name fields.
   Returns a value between 0.00 and
   1.00, with 1.00 being an exact-fit
*/
double NH_calc_score(   SegList qSegs, int numQSegs,           SegList
t evalSegs, int numEvalSegs,           SegList
tVariants querySegmentVariants,       char
                                     *primaryCulture,           char
                                     *secondaryCulture,          NHComp
Parms *compParms,                                     NHName
Parms *nameParms,                                     NHName
Fields nameField,                                     char
*origQNameField,                                     char
*origEvalNameField,                                  int
*numSegsScored,                                      double
*bestSegScores)
{
    NHAnchorSegMode    anchorSeg;
    NHSegScoreMode     scoreMode;
    double             oopsFactor;
    double             absDeltaTAQFactor;
    double             absDisTAQFactor;
    double             delTAQFactor;
    double             disTAQFactor;
    bool               matchInit;
    double             initScore;
    double             initialOnInitialMatchScore;
re;
    bool               checkVariant;
    // double          variantScore;
    bool               leftDigraphBias;
    double             anchorFactor;
    double             nameUnknownScore;
    double             noNameScore;
    double             scoresTable[NH_MAX_SEGS_AFTER_T
TAQ][NH_MAX_SEGS_AFTER_TAQ]; // scores for segment pairs
    int               qIndex;
    // temp index for query segments

```



```

        int                                evalIndex; //
temp index for eval segments
        int                                qSegLen;
//    hold string length of query segment
        int                                evalSegLen; //
hold string length of eval segment
        double                                diScore =
0.0;    //    temp score for single pair comparison
        double                                hiScore =
0.0;    //    temp score to hold best score as we iterate,

                                                //    which lets us avoid
best_score in mode=BEST
        bool                                areVariants;
//    temp flag to hold if the pair are variants
        double                                returnValue = 0.0;
        NHVariantTable                        *variantTable;
        double                                varScore;
        NHVarId                                evalSegVarId
;
        bool                                scoreTags;
        double                                compressedNameScore;
        bool                                checkCompressedName;

        //    set some paramters based on the name field
if (nameField == NH_LAST_NAME) {
    anchorSeg = compParms->getSnAnchorSegmentMode();
    scoreMode = compParms->getSnSegmentScoreMode();
    oopsFactor = compParms->getSnOOPSFactor();
    matchInit = compParms->getMatchSnInitial();
    initScore = compParms->getSnInitialScore();
    initialOnInitialMatchScore = compParms-
>getSnInitialOnInitialMatchScore();
    checkVariant = compParms->getUseSnVariants();
    anchorFactor = compParms->getSnAnchorFactor();
    leftDigraphBias = compParms->getUseSnLeftBias();
    nameUnknownScore = compParms->getLNUScore();
    noNameScore = compParms->getNLNScore();
    scoreTags = compParms->getScoreSnTAQs();
    absDelTAQFactor = compParms->getAbsDelSnTAQFactor();
    absDisTAQFactor = compParms->getAbsDisSnTAQFactor();
    delTAQFactor = compParms->getDelSnTAQFactor();
    disTAQFactor = compParms->getDisSnTAQFactor();
    compressedNameScore = compParms->getSnCompressedNameScore();
    checkCompressedName = compParms->getCheckSnCompressedName();
    variantTable = nameParms->snVariantTable;
}
else {
    anchorSeg = compParms->getGnAnchorSegmentMode();
    scoreMode = compParms->getGnSegmentScoreMode();
    oopsFactor = compParms->getGnOOPSFactor();
    matchInit = compParms->getMatchGnInitial();
    initScore = compParms->getGnInitialScore();
    initialOnInitialMatchScore = compParms-
>getGnInitialOnInitialMatchScore();
    checkVariant = compParms->getUseGnVariants();
    anchorFactor = compParms->getGnAnchorFactor();
    leftDigraphBias = compParms->getUseGnLeftBias();
    nameUnknownScore = compParms->getFNUScore();
    noNameScore = compParms->getNFNScore();

```

```

        scoreTags = compParms->getScoreGnTAQs();
        absDelTAQFactor = compParms->getAbsDelGnTAQFactor();
        absDisTAQFactor = compParms->getAbsDisGnTAQFactor();
        delTAQFactor = compParms->getDelGnTAQFactor();
        disTAQFactor = compParms->getDisGnTAQFactor();
        compressedNameScore = compParms->getGnCompressedNameScore();
        checkCompressedName = compParms->getCheckGnCompressedName();
        variantTable = nameParms->gnVariantTable;
    }

    // clear out the scores table
    for (qIndex = 0; qIndex < NH_MAX_SEGS_AFTER_TAO; ++qIndex)
        for (evalIndex = 0; evalIndex < NH_MAX_SEGS_AFTER_TAO; ++evalIndex)
            scoresTable[qIndex][evalIndex] = 0.0;

    // now go through each possible combination of segment pairs
    // (created by matching a query segment against an eval
    segment).
    // Store the scores in the scoresTable.
    for (qIndex = 0; qIndex < numQSegs; ++qIndex) {

        qSegLen = strlen(qSegs[qIndex].segString);

        for (evalIndex = 0; evalIndex < numEvalSegs; ++evalIndex) {

            evalSegLen = strlen(evalSegs[evalIndex].segString);

            // first check for either the query or eval segment
            being
            // blank.
            if ((qSegLen == 0) || (evalSegLen == 0)) {
                // We make a distinction between "unknown"
                // and "none". The table below shows the
                scores
                // we assign for the various combinations of
                Known - K,
                // Unknown - U, and None -N.
                //
                //          |      K
                //          |      U
                //          |      N
                // -----
                //          K      |      N/A
                |      unknownScore          |      NoneScore
                // -----
                //          U      |      unknownScore |      (unknownScore
                e + 1) / 2 |      (unknownScore + 1) / 2
                // -----
                //          N      |      NoneScore      |      (unkno
                wnScore + 1) / 2 |      (NoneScore + 1) / 2
                // -----

            if (qSegs[qIndex].status ==
            NH_NAME_FIELD_STATUS_KNOWN) {
                // we should not need to check for both
                being known

```

```

                                if (evalSegs[evalIndex].status ==
NH_NAME_FIELD_STATUS_UNKNOWN)
                                diScore = nameUnknownScore;
                                else // must be
NH_NAME_FIELD_STATUS_NON_EXISTANT
                                diScore = noNameScore;
                                }
                                else if (qSegs[qIndex].status ==
NH_NAME_FIELD_STATUS_UNKNOWN) {
NH_NAME_FIELD_STATUS_UNKNOWN)
                                if (evalSegs[evalIndex].status ==
NH_NAME_FIELD_STATUS_UNKNOWN)
                                diScore = nameUnknownScore;
                                else if (evalSegs[evalIndex].status ==
NH_NAME_FIELD_STATUS_UNKNOWN)
                                diScore = (nameUnknownScore + 1.0) /
2.0;
                                else // must be
NH_NAME_FIELD_STATUS_NON_EXISTANT, same score as
                                // above, but we
repeat it in case we change behavior later
                                diScore = (nameUnknownScore + 1.0) /
2.0;
                                }
                                else { // query must be
NH_NAME_FIELD_STATUS_NON_EXISTANT)
                                if (evalSegs[evalIndex].status ==
NH_NAME_FIELD_STATUS_UNKNOWN)
                                diScore = noNameScore;
                                else if (evalSegs[evalIndex].status ==
NH_NAME_FIELD_STATUS_UNKNOWN)
                                diScore = (nameUnknownScore + 1.0) /
2.0;
                                else // must be
NH_NAME_FIELD_STATUS_NON_EXISTANT, same score as
                                // above, but we
repeat it in case we change behavior later
                                diScore = (noNameScore + 1.0) / 2.0;
                                }
                                }
                                else {
                                // check the variants if
                                // - we are supposed to
                                // - we have a list of variants to
check
                                // - there is a variant for this
segment of the query
                                // Note we must check the secondary
variants if the
                                // primary check does not find a
variant.
                                areVariants = false;
                                if (checkVariant && (querySegmentVariants !=
NULL) &&
                                (querySegmentVariants[qIndex] !=
NULL)) {
                                // so see if the eval name segment has
any variants in the variant table
                                evalSegVarId = variantTable-
>getVariantIdForName(evalSegs[evalIndex].segString);
                                if (evalSegVarId !=
NH_VAR_NOT_FOUND) {

```

```

// yes, it did have some
variants, so see if there is an intersection
varScore =
querySegmentVariants[qIndex]-
>getVariantScoreForIdAndCulture(evalSegVarId, primaryCulture);
if (varScore !=
NH_VARIANTS_NOT_RELATED) {
    areVariants = true;
    diScore = varScore;
}
else {
    // variants were not
related, so check for the secondary
    // variant source
    // Put a check in here to
see if the primary culture
    // code was
NH_CULTURE_CODE_GENERIC. If so, we can skip this check
    // since the secondary code
is always generic
    if (strcmp(primaryCulture,
NH_CULTURE_CODE_GENERIC)) {
        varScore =
querySegmentVariants[qIndex]-
>getVariantScoreForIdAndCulture(evalSegVarId, secondaryCulture);
        if (varScore !=
NH_VARIANTS_NOT_RELATED) {
            areVariants =
true;
            diScore =
varScore;
        }
    }
}

// now, if we did not find variants above,
check for initials
// do we have an initial and are we supposed to
check them?
if (areVariants == false) {
    if (matchInit && (qSegLen == 1 ||
evalSegLen == 1)) {
        // does the first char match ?
        if (qSegs[qIndex].segString[0] ==
evalSegs[evalIndex].segString[0]) {
            // if the second char
matches, we have an initial on initial match,
            // since we know the length
of atleast one of them is 1.
            if (qSegs[qIndex].segString[1]
== evalSegs[evalIndex].segString[1])
                diScore =
initialOnInitialMatchScore;
            else // initial
match, but one was more than a single character
                diScore =
initScore; // so assign initScore
        }
    }
}
else

```

```

                                diScore =
0.0;                          // no match at all, since first char was off
                                }
                                else {                          // else not initials
or we shouldn't check them
                                // when here, we do not
have unknowns, variants, or initials,
                                // so do a digraph
comparison.
                                diScore =
NH_digraph_score(qSegs[qIndex].segString, qSegLen,
evalSegs[evalIndex].segString, evalSegLen,
                                leftDigraphBias);
                                }
                                // end, if (areVariants == false)
                                // end, else, both segs are known
(neither name is blank)
/*
Here we need to handle the oops and anchor
segment parameters.
oops specifies a factor to multiply by the score
when the segments
are not in the same position.
AnchorSeg, AnchorFactor specify a factor to
multiply matches that
are in the same segment position, but are in a
segment other than
the stated AnchorSeg. Note that AnchorSeg does
not get applied in
average mode, because otherwise a 2 segment name
that was
an exact match would get less than 1.0, since
the segment that
was not in the anchor segment would be
penalized. Anchor Factor
is meant more to provide a penalty when a
(relatively)
unimportant segment is used as the sole
contributor to
the score.

Note that only one of the factors may be
applied, since oops only
gets applied to segments that are out of place,
and anchorFactor
only gets applied to matches that are in place.

AnchorSeg is also used to determine segment
alignment. anchorSeg
value 1 indicates segments should be lined up on
the left, while
value 2 indicates they should be lined up on the
right. A value
of 0 indicates they should be lined up on the
left (this is the
default.
*/

```

```

        switch (anchorSeg) {
            case 0
:
anchor segment designation // no
                                if (qIndex != evalIndex) // out of
place, so apply oops
                                diScore *= oopsFactor;
                                break;
            case 1
:
segment is most important // first
                                if (qIndex !=
evalIndex) // out of place, so apply oops
                                diScore *= oopsFactor;
                                else
                                if ((qIndex != 0) && (scoreMode !=
NH_SEGMODE_AVG)) // if not the first segment (anchor seg)
                                diScore *=
anchorFactor; // apply the anchorFactor, so long as the
break;
                                // scoreMode is not
NH_SEGMODE_AVG
            case 2 : /* If not last-to-last match... */
                                if ((qIndex == numQSegs - 1) && (evalIndex
== numEvalSegs - 1))
                                ; // no modification, since both are
end segments
                                else {
                                // see if they are in the same
position, counting back from the end
                                if ((numQSegs - qIndex) ==
(numEvalSegs - evalIndex))
                                if (scoreMode !=
NH_SEGMODE_AVG) // skip anchor factor in average seg mode
                                diScore *=
anchorFactor; // apply the anchorFactor
                                else
                                diScore *= oopsFactor;
                                }
                                break;
        }

// Now we need to apply the TAQ values to the
score,
// but only if they wanted to, and we have a score
// greater than 0 (otherwise, factors have no
effect).
        if ((scoreTaq) && (diScore > 0.0))
            NH_apply_TAQs_to_score(&diScore, &qSegs[qIndex],
&evalSegs[evalIndex],

            absDelTAQFactor, absDisTAQFactor,

            delTAQFactor, distTAQFactor);

        if (numQSegs > numEvalSegs) // always store
smaller dimension as rows
            scoresTable[evalIndex][qIndex] = diScore;
        else
            scoresTable[qIndex][evalIndex] = diScore;

```

```

        hiScore = hiScore > diScore ? hiScore : diScore;
    } // for evalIndex

} // for qIndex

// now figure out a composite score from all the best scores
// Note that for Best score, we must set the number of segments
// that were scored, and fill an array containing those scores
-
// these will be used later to sort hits).
// The exception to this is when either the query or the
// eval name field has just 1 segment, in which case we only
// score one segment, which becomes the score (in all modes).
if ((numEvalSegs == 1) || (numQSegs == 1)) {
    if (scoreMode == NH_SEGMODE_HIGHEST) {
        *numSegsScored = 1; // note that we only
scored 1 segment
        bestSegScores[0] = hiScore; // save the
singly scored segment
    }
    returnValue = hiScore;
}
else {
    // both have more than 1 segment
    if (numQSegs > numEvalSegs) { // always call
functions with smaller dimension as rows
        if (scoreMode == NH_SEGMODE_HIGHEST) {
            NH_best_score_for_highest_mode(numEvalSegs,
numQSegs, hiScore, bestSegScores, scoresTable);
            *numSegsScored = numEvalSegs; // note
that we only scored numEvalSegs segments
            returnValue = hiScore;
        }
        else
            returnValue = NH_best_score(numEvalSegs,
numQSegs, scoreMode, scoresTable);
    }
    else {
        if (scoreMode == NH_SEGMODE_HIGHEST) {
            NH_best_score_for_highest_mode(numQSegs,
numEvalSegs, hiScore, bestSegScores, scoresTable);
            *numSegsScored = numQSegs; // note
that we only scored numQSegs segments
            returnValue = hiScore;
        }
        else
            returnValue = NH_best_score(numQSegs,
numEvalSegs, scoreMode, scoresTable);
    }
}

// here we need to see if we are supposed to check compressed
names.
// if so, we have to call the NH_check_compressed_name()
function.
// If that function returns true, we pick the higher of the
// compressedScore (which is a parameter) and the current
returnValue.

```

```

        if (checkCompressedName &&
            NH_check_compressed_name(origQNameField,
origEvalNameField,

                                nameParms->getSegmentBreakChars(),

                                nameParms->getNoiseChars()))
            returnValue = returnValue > compressedNameScore ?
returnValue : compressedNameScore;

        return returnValue;
    } /* NH_calc_score */

/* NH_check_compressed_name

    Compresses both names passed in, and sees if they are exact
    matches.

    The compression is implemented by skipping characters specified in
    compressChars.
*/
bool NH_check_compressed_name(char *qSegString, char *evalSegString,
char *compressCharsPart1,

                                char *compressCharsPart2)
{
    char compressedQuerySeg[NH_MAX_SEG_LENGTH + 1];
    char compressedEvalSeg[NH_MAX_SEG_LENGTH + 1];
    char compressChars[200 + 1];
    char *p;
    char *q;

    // first, combine the compressCharsPart1 and compressCharsPart1
strings
    strcpy(compressChars, compressCharsPart1);
    strcat(compressChars, compressCharsPart2);

    // compress the query segment
    for (p = qSegString, q = compressedQuerySeg; *p != EOS; p++)
        if (strchr(compressChars, *p) == NULL)
            *q++ = *p;
    *q = EOS;

    // compress the query segment
    for (p = evalSegString, q = compressedEvalSeg; *p != EOS; p++)
        if (strchr(compressChars, *p) == NULL)
            *q++ = *p;
    *q = EOS;

    // at this point, we are not necessarily upper cased, so ignore
case
    // during the string copy
    return !strcasecmp(compressedQuerySeg, compressedEvalSeg);
} /* NH_check_compressed_name */

```



```

/* NH_best_score
    From a matrix of scores compute the highest possible
combination    of scores. During the evaluation of the matrix, a given row
or              column must provide one and only one score.

                We use a mode to determine how we calculate a score. The
mode            can be either NH_SEGMODE_AVG or NH_SEGMODE_LOWEST. Both
modes          start out by selecting the combination of values (with no
row or        column being used more than once) that gives the highest
sum. Then,    for mode = NH_SEGMODE_AVG, the final score is the average of
all           these scores. For NH_SEGMODE_LOWEST, it is the worst of
these scores.

                If the matrix is non-square (x <> y), our final score
calculation    only includes N values, where N is the lesser dimension. We
still          use all the possible squares in the matrix to perform our
selection,     but the final score does not consider part of the matrix.

                To perform the work, we figure out which type of matrix we
are            dealing with (the dimensions). We use that to select an
array that     contains
contains       the column indexes for each valid combination of segments
(where        valid means no column participates twice).

                Our matrix always comes either as a square, or as a fat,
short matrix.  That is, the number of rows is always less than or equal to
the number of  columns. This way, we do not have to specify as many
combination   arrays,
since we only have to cover a 2 X 3 array, and not a 3 X 2.

                Also, before this function, we see if either name has just 1
segment, in which case we use the best score.
*/
double        NH_best_score(int xDim, int yDim, NHSegScoreMode scoreMode,
                                                                    double
scores[NH_MAX_SEGS_AFTER_TAO][NH_MAX_SEGS_AFTER_TAO])
{
    byte *comboIndexesPtr;          // points to array that
holds valid column index combos
    int    numCominations;

    switch (xDim) {
        case 2:
            switch (yDim) {
                case 2: // 2 by 2
                    comboIndexesPtr = twoByTwo;

```

```

        numCominations = 2;
        break;
    case 3:          // 2 by 3
        comboIndexesPtr = twoByThree;
        numCominations = 6;
        break;
    case 4:          // 2 by 4
        comboIndexesPtr = twoByFour;
        numCominations = 12;
        break;
    case 5:          // 2 by 5
        comboIndexesPtr = twoByFive;
        numCominations = 20;
        break;
    default:         // must be greater than 5,
so just use first five
        comboIndexesPtr = twoByFive;
        numCominations = 20;
        break;
    }
    break;
case 3:
    switch (yDim)    {
        case 3:          // 3 by 3
            comboIndexesPtr = threeByThree;
            numCominations = 6;
            break;
        case 4:          // 3 by 4
            comboIndexesPtr = threeByFour;
            numCominations = 24;
            break;
        case 5:          // 3 by 5
            comboIndexesPtr = threeByFive;
            numCominations = 60;
            break;
        default:         // must be greater than 5,
so just use first five
            comboIndexesPtr = threeByFive;
            numCominations = 60;
            break;
    }
    break;
case 4:
    switch (yDim)    {
        case 4:          // 4 by 4
            comboIndexesPtr = fourByFour;
            numCominations = 24;
            break;
        case 5:          // 4 by 5
            comboIndexesPtr = fourByFive;
            numCominations = 120;
            break;
        default:         // must be greater than 5,
so just use first five
            comboIndexesPtr = fourByFive;
            numCominations = 120;
            break;
    }
    break;
case 5:
    switch (yDim)    {

```

```

        case 5:                //      5 by 5
            comboIndexesPtr = fiveByFive;
            numCominations = 120;
            break;
        default:                //      must be greater than 5,
so just use first five
            comboIndexesPtr = fiveByFive;
            numCominations = 120;
            break;
    }
    break;
    default:                    //      must be greater than 5, so just use
first five
                                //      also, since xDim
is <= yDim, we do not have to
                                //      handle 5 X 2, 5 X
3, etc
                                comboIndexesPtr = fiveByFive;
                                numCominations = 120;
                                break;
    }

    //      we always use xDim matrix cells to compute the score, since
it
    //      is the smaller of the dimensions. We go through each
combination
    //      and evaluate the scores found in the scores array for the
    //      particular combination of indexes.
    //      Each evaluation must consider xDim values, so each pass
through the
    //      loop collects xDim values.
    //      The values from the comboIndexesPtr array are the column
indexes.
    //      numCominations is the number of times we iterate through the
loop to
    //      look at a combination of elements in the score matrix.
    //
    //      For example:
    //      if I have a 2 X 3 matrix, I need to find the best valid 2
segment
    //      combination (since 2 is xDim). There are 6 possible
combinations,
    //      and the column values are stored as pairs in the twoByThree
array.
    //      The row values are implicitly 0 and 1 for each pair, so I
end up
    //      checking:
    //
    //      scores[0][twoByThree[0]] +
scores[1][twoByThree[1]];
    //      scores[0][twoByThree[2]] +
scores[1][twoByThree[3]];
    //      scores[0][twoByThree[4]] +
scores[1][twoByThree[5]];
    //      scores[0][twoByThree[6]] +
scores[1][twoByThree[7]];
    //      scores[0][twoByThree[8]] +
scores[1][twoByThree[9]];
    //      scores[0][twoByThree[10]] +
scores[1][twoByThree[11]];
    //
    double      tempScoreTotal;

```

```

double    tempLowScore;
double    tempVal;
double    highestTotal = 0.0;
double    bestLowScore = 0.0;
int       comboArrayIndex = 0;
int       i, row;

for (i = 0; i < numCominations; i++) {
    tempScoreTotal = 0.0;
    tempLowScore = 1.0;
    for (row = 0; row < xDim; row++) {
        // get a single score
        tempVal =
scores[row][comboIndexesPtr[comboArrayIndex]];
        // now see if score is the low score for this combo
        if (tempVal < tempLowScore)
            tempLowScore = tempVal;

        // include this cell in the total for this
combination
        tempScoreTotal += tempVal;

        // look at next item in the combination (or the
next combination)
        comboArrayIndex++;
    }
    // see if the low score is better than our previous low
score
    if (tempLowScore > bestLowScore)
        bestLowScore = tempLowScore;
    // see if this score is higher than our previous highest
    if (tempScoreTotal > highestTotal)
        highestTotal = tempScoreTotal;
}
if (scoreMode == NH_SEGMODE_AVG)
    return highestTotal / xDim;
else
    return bestLowScore;
}

```

/\* NH\_best\_score\_for\_highest\_mode

This is a special version of NH\_best\_score. For a complete description of how the combination stuff works, see the comments

for NH\_best\_score.

We made this a separate function because:

- it has to return (by reference) an array of scores. The other
- modes only have to return a score for the name.
- The way we figure out which array of scores to return is
- much more involved than NH\_best\_score.
- Since we only do this stuff in highest mode, we did not
- want to slow down the processing of NH\_best\_score by passing
- extra parameters and adding lots of "if" statements.

This function was added so that we can figure out which combination of segments gives us the highest scores, with the highest score being most important, the next highest score being the second most important, etc. Note that this is different from average score, where we are looking for the highest sum of scores. In that case, the highest score is no more important than the lowest score. In fact, the combination chosen in average mode might not even include the single highest segment score.

To achieve our goal, we evaluate each possible combination of index pairings. Each combination gives us an array of N scores, where n is the smaller dimension in the matrix.

We sort each combination so that the highest score appears first in the array, and so on. If this is the first combination we have evaluated, it becomes the one to beat, so we fill up the array of scores we were passed by reference with this array of scores. We then go through the rest of the combinations looking for an array that beats the current one to beat. To beat it, as we walk through the array, we compare the scores from each array. If they are equal, we move on to the next one. Otherwise, the higher score wins.

To help speed things up, we get passed in the high score, which is the high score of the entire matrix (note this high score could appear more than once). We use this high score to quickly discount combinations as not being possible contenders. If, after sorting a contender array, the first item is not the high score we were passed, this combination could not possibly be the one, so why bother copying all the array elements?

Note that we check before entering this function to make sure both dimensions are bigger than 1. And we make sure that xdim is the smaller of the dimensions (or they are equal).

```

*/
void NH_best_score_for_highest_mode(int xDim, int yDim, double
highestScore,
double
*bestSegScores,
  
```

```

double
scores[NH_MAX_SEGS_AFTER_TAO][NH_MAX_SEGS_AFTER_TAO])
{
    byte *comboIndexesPtr;           // points to array that
    holds valid column index combos
    int    numCominations;

    switch (xDim) {
        case 2:
            switch (yDim) {
                case 2: // 2 by 2
                    comboIndexesPtr = twoByTwo;
                    numCominations = 2;
                    break;
                case 3: // 2 by 3
                    comboIndexesPtr = twoByThree;
                    numCominations = 6;
                    break;
                case 4: // 2 by 4
                    comboIndexesPtr = twoByFour;
                    numCominations = 12;
                    break;
                case 5: // 2 by 5
                    comboIndexesPtr = twoByFive;
                    numCominations = 20;
                    break;
                default: // must be greater than 5,
                    so just use first five
                    comboIndexesPtr = twoByFive;
                    numCominations = 20;
                    break;
            }
            break;
        case 3:
            switch (yDim) {
                case 3: // 3 by 3
                    comboIndexesPtr = threeByThree;
                    numCominations = 6;
                    break;
                case 4: // 3 by 4
                    comboIndexesPtr = threeByFour;
                    numCominations = 24;
                    break;
                case 5: // 3 by 5
                    comboIndexesPtr = threeByFive;
                    numCominations = 60;
                    break;
                default: // must be greater than 5,
                    so just use first five
                    comboIndexesPtr = threeByFive;
                    numCominations = 60;
                    break;
            }
            break;
        case 4:
            switch (yDim) {
                case 4: // 4 by 4
                    comboIndexesPtr = fourByFour;
                    numCominations = 24;
                    break;
                case 5: // 4 by 5

```

```

        comboIndexesPtr = fourByFive;
        numCominations = 120;
        break;
    default:          // must be greater than 5,
so just use first five
        comboIndexesPtr = fourByFive;
        numCominations = 120;
        break;
    }
    break;
case 5:
    switch (yDim)      {
        case 5:        // 5 by 5
            comboIndexesPtr = fiveByFive;
            numCominations = 120;
            break;
        default:       // must be greater than 5,
so just use first five
            comboIndexesPtr = fiveByFive;
            numCominations = 120;
            break;
    }
    break;
default:          // must be greater than 5, so just use
first five
// also, since xDim
is <= yDim, we do not have to
// handle 5 X 2, 5 X
3, etc
        comboIndexesPtr = fiveByFive;
        numCominations = 120;
        break;
    }

    // we always use xDim matrix cells to compute the score, since
it
    // is the smaller of the dimensions. We go through each
combination
    // and evaluate the scores found in the scores array for the
    // particular combination of indexes.
    // Each evaluation must consider xDim values, so each pass
through the
    // loop collects xDim values.
    // The values from the comboIndexesPtr array are the column
indexes.
    // numCominations is the number of times we iterate through the
loop to
    // look at a combination of elements in the score matrix.
    //
    // For example:
    // if I have a 2 X 3 matrix, I need to find the best valid 2
segment
    // combination (since 2 is xDim). There are 6 possible
combinations,
    // and the column values are stored as pairs in the twoByThree
array.
    // The row values are implicitly 0 and 1 for each pair, so I
end up
    // checking:
    //
    // scores[0][twoByThree[0]] +
scores[1][twoByThree[1]];

```

```

//      scores[0][twoByThree[2]] +
scores[1][twoByThree[3]];
//      scores[0][twoByThree[4]] +
scores[1][twoByThree[5]];
//      scores[0][twoByThree[6]] +
scores[1][twoByThree[7]];
//      scores[0][twoByThree[8]] +
scores[1][twoByThree[9]];
//      scores[0][twoByThree[10]] +
scores[1][twoByThree[11]];
//
double      tempSegScores[NH_MAX_SEGS_AFTER_TAO];
int          comboArrayIndex = 0;
int          i, row;
bool         includesHighestScore;
double       swapVal;
int          tempIndex;
double       compVal;
int          numChanges;
double       tempVal;

//      init the temp seg scores array to zeros, so that the first
//      comparison will fail.
for (tempIndex = 0; tempIndex < xDim; tempIndex++) {
    bestSegScores[tempIndex] = 0;
}

for (i = 0; i < numCominations; i++) {
    includesHighestScore = false; //      assume this combo does
not

//      include the best score
for (row = 0; row < xDim; row++) {
    //      get a single score
    tempVal =
scores[row][comboIndexesPtr[comboArrayIndex]];

//      now see if score is the low score for this combo
if (tempVal == highestScore)
    includesHighestScore = true;

//      save this value as part of our temp array of
scores
//      that we will sort below
tempSegScores[row] = tempVal;

//      look at next item in the combination (or the
next combination)
    comboArrayIndex++;
}
//      see if this combo includes the best score.  If so,
sort it
//      and then compare it to the current numbers in
bestSegScores.
if (includesHighestScore == true) {
    //      sort the numbers in bestSegScores
    while (1) {
        numChanges = 0;
        for (tempIndex = 1; tempIndex < xDim;
tempIndex++) {

```



```

tempSegScores[tempIndex]    if (tempSegScores[tempIndex - 1] <
{
    swapVal = tempSegScores[tempIndex -
1]);
    tempSegScores[tempIndex - 1] =
tempSegScores[tempIndex];
    tempSegScores[tempIndex] = swapVal;
    numChanges++;
}
}
if (numChanges == 0)
    break;
}

// now compare these temp scores to the current
best scores
for (tempIndex = 0; tempIndex < xDim;
tempIndex++)
{
    compVal = tempSegScores[tempIndex] -
bestSegScores[tempIndex];
    if (compVal > 0) {
        // temp scores are better, so replace
the best scores with them
        for (tempIndex = 0; tempIndex < xDim;
tempIndex++)
        {
            bestSegScores[tempIndex] =
tempSegScores[tempIndex];
        }
        break;
    }
    else
    if (compVal < 0) {
        // current scores are better, so
break out
        break;
    }
    // otherwise, just continue the loop.
}
}
}
}
}

```

/\* digraph\_score

This is the core of the name-check algorithm.

A value from 0.0 to 1.0 is calculated based on the number of digraphs which match between the two given strings.

A bias can be used so that digraphs on the right end of the strings count less than those on the left.

Notes:

The routine ensures that a digraph can only participate in a match once.

Each match results in two points being added to the total.

The

final score is the total number of points divided by the

number

of digraphs that could have matched.

digraph The bias works by discounting the score we award for a  
the match. As we move into the segment, we subtract 0.1 from  
the match score.

normally The weight table is used to adjust the divisor (which is  
the case the total number of digraphs that could have matched). In  
exact match of bias, we need to decrease that number. Otherwise, an  
from the would not return a 1.0, since we would only be deducting  
table factors score (the numerator), and not the divisor. The weight  
match for correspond to the score that would be assigned to an exact  
add .9, then each possible length. In other words, we start at 1, then  
add .8, etc. (the same pattern we use to deduct from the  
match score)

```

*/
double    NH_digraph_score(char *qSeg, int qSegLen,
char *evalSeg, int evalSegLen,
                                bool useLeftDigraphBias)
{
    char tempDigraphStr[2 + 1]; // storage for a digraph string

    // terminate the temp digraph string once
    tempDigraphStr[2] = EOS;

    // These are the weights a name has when using a biased
    // (left-to-right) calculation. They end up being used as the
denominator
    // for the final score calculation
    static const double NH_dig_bias_weights[NH_MAX_SEG_LENGTH + 2]
        = { 1.0, 1.0, 1.9, 2.7, 3.4, 4.0, 4.5, 4.9, 5.2, 5.4, 5.5,
5.6, 5.7,
                                5.8, 5.9, 6.0, 6.1, 6.2, 6.3, 6.4,
6.5, 6.6, 6.7, 6.8, 6.9, 7.0,
                                7.1, 7.2, 7.3, 7.4, 7.5, 7.6};

    // an array of 'Y' or 'N' values, one for each possible digraph
    // position in the eval segment. Each starts out at 'N' and
gets
    // to 'Y' when (and if) it gets used.
    // Note that we must add 1 because we normally pad the name
with
    // spaces.
    char alreadyMatched[NH_MAX_SEG_LENGTH + 1]; // max digraphs =
NAME_SIZE + 1

    // Forget all previous matches.
    memset(alreadyMatched, 'N', sizeof alreadyMatched);

    // Now count the number of elements involved in matching.
double    qBiasFactor = 0.9; // 0.9 because

```

```

of leading digraph check
    double    evalBiasFactor = 0.9;           //    see note below
    double    matchPoints;
    char      *evalSegString;

    //    start out by checking the first character, which is a
special    //    case.  It forms an implied digraph of " X" (space, followed
by        //    the character.  Thus, if both the query and eval have the
same      //    first character, we give them 2 match points.
    //    Also, since we really start our loop with the second
digraph,  //    we set the bias factors to 0.9 rather than 1.0
    if (qSeg[0] == evalSeg[0])    {
        matchPoints = 2.0;
    }
    else
        matchPoints = 0.0;

    for (int queryIndex = 0; queryIndex < qSegLen - 1; ++queryIndex) {
        /* see if this digraph occurs in database name */
        tempDigraphStr[0] = qSeg[queryIndex];
        tempDigraphStr[1] = qSeg[queryIndex + 1];
        evalSegString = evalSeg;
        if (useLeftDigraphBias) {
            //    bring down the query bias by 0.1 each time,
until we get to 0.1
            if ((queryIndex > 0) && (queryIndex < 10))
                qBiasFactor -= 0.1;
        }
        do {
            evalSegString = strstr(evalSegString, tempDigraphStr);
            if (evalSegString != NULL) {
                int evalMatchOffset = evalSegString - evalSeg;

                if (alreadyMatched[evalMatchOffset] == 'N') {
                    alreadyMatched[evalMatchOffset] = 'Y';
                    if (useLeftDigraphBias) { /* decrement
eval match-bias, minimum 0.10 */
                        evalBiasFactor = 1.0 - 0.1 *
(evalMatchOffset + 1);
                        if (evalBiasFactor < 0.1)
                            evalBiasFactor = 0.1;
                        matchPoints += qBiasFactor +
evalBiasFactor;
                    }
                    else
                        matchPoints += 2.0;
                    break;
                }
                else
                    evalSegString++;
            }
        } while (evalSegString != NULL);
    }

    //    now do a check for the "hidden" digraph at the end of the
segment

```

```

//      to account for the non-existent trailing space
if (qSeg[qSegLen - 1] == evalSeg[evalSegLen - 1]) {
    if (useLeftDigraphBias) {
        evalBiasFactor = 1.0 - 0.1 * evalSegLen;
        if (evalBiasFactor < 0.1)
            evalBiasFactor = 0.1;
        //      don't forget to bring down the query bias by 0.1
also,
        //      unless we are at 0.1
        if ((queryIndex > 0) && (queryIndex < 10))
            qBiasFactor -= 0.1;
        matchPoints += qBiasFactor + evalBiasFactor;
    }
    else
        matchPoints += 2.0;
}

```

```

// The return value is the number of elements involved in matching
// compared to the total number of elements.
return useLeftDigraphBias
    ? matchPoints /
      (NH_dig_bias_weights[qSegLen + 1] + NH_dig_bias_weights[evalSegLen + 1])
    : matchPoints / (qSegLen + evalSegLen +
2);
} /* NH_digraph_score */

```

/\*  
This function adjusts the diScore (which already has some value)  
based  
on the TAQ values that are attached to the two segments passed in.

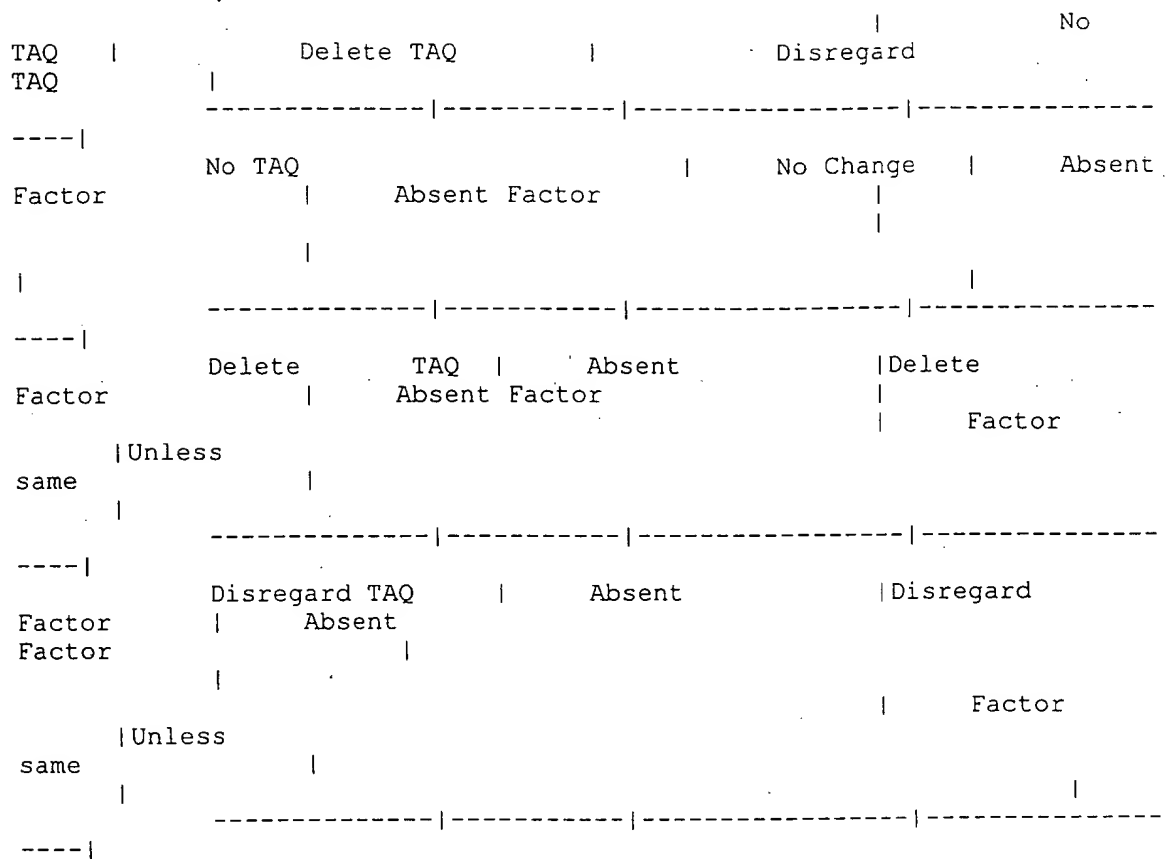
In the NameHunter system, TAQs are broken up into two types  
(disregard and  
delete). In general, disregard TAQs (e.g. "Jr.") contain more  
meaningful information than delete TAQs (e.g. "Mr."), and thus  
disregard TAQs are considered more important when  
evaluating/comparing  
TAQs between segments.

There are three factors involved in modifying the score. These  
are

- delete factor
- disregard factor
- absent factor

When applied, a factor is multiplied by the existing score.  
However,  
deciding which factor (if any) to apply is somewhat complex,  
especially  
when one or both of the segments have multiple TAQ values. For  
this  
reason, we describe the multi-TAQ situation separately.

For situations where both segments have either 0 or 1 TAQ values,  
we  
use the following matrix to choose a factor to apply:



For the multiple case, we use the algorithm below. A general word about the alg - we are treating disregard as more important than delete, so we start out by checking for disregards. All it takes is for one disregard value in each of the segments to match to avoid applying the disregard factor. The same goes for deletes. If we have any dis values in one segment, but none in the other, we apply the absent factor.

Assuming segments S1 and S2:

```

- Look for dis segments in S1
- if found
  - if same segment found in S2
    - go on to delete processing
  - else
    - if no dis segments in S2
      - apply absent value
    - else, continue looking for dis segments in
      S1 that match S2
      if we get to end of S1 segments and still
      have not found a
        matching dis in S2, apply dis factor.
  - else (no dis found in S1)
    - look for dis in S2
      - if found
        - apply absent

```

```

- else
- go on to delete processing

```

Delete processing:

```

- look for deletes in S1
- if found
- if same seg found in S2
- do nothing
- else
- if no deletes in S2
- apply absent
- else
- continue to look for deletes in S1.

```

If we get to end if

S1 segments and do not find any

deletes that match a

delete in S2, apply delete factor

```

- else (not deletes found in S1)
- look for deletes in S2
- if delete found
- apply absent
- else
- do nothing.

```

\*/

```

void NH_apply_TAQs_to_score(double *diScore, Segment *qSeg, Segment
*evalSeg,

```

```

double absDelTAQFactor,

```

```

double absDisTAQFactor,

```

```

double delTAQFactor,

```

```

double distTAQFactor)

```

```

{

```

```

int numQTAQs = qSeg->numTAQs;

```

```

int numEvalTAQs = evalSeg->numTAQs;

```

```

double applyFactor = 1.0;

```

```

// handle the simple case first

```

```

if ((numQTAQs <= 1) && (numEvalTAQs <= 1)) {

```

```

switch (numQTAQs) {

```

```

case 0:

```

```

if (numEvalTAQs == 1) {

```

```

if (evalSeg->taqList[0].taqAction ==

```

```

NH_TAQ_ACTION_DELETE)

```

```

applyFactor = absDelTAQFactor;

```

```

else

```

```

applyFactor = absDisTAQFactor;

```

```

}

```

```

break;

```

```

case 1:

```

```

if (numEvalTAQs == 1) {

```

```

// both segs have 1 TAQ value, so

```

```

// figure out the type of TAQs involved

```

```

if (qSeg->taqList[0].taqAction ==

```

```

NH_TAQ_ACTION_DELETE) {

```

```

if (evalSeg->taqList[0].taqAction ==

```

```

NH_TAQ_ACTION_DELETE) {
    // same action, so see if
    string are the same
    if (strcmp(qSeg-
>taqList[0].segString,
    evalSeg->taqList[0].segString))
        applyFactor =
delTAQFactor; // they were different, so apply delete
factor
    }
    else // not the same
        action, so do the absent
        applyFactor = absDisTAQFactor;
    }
    else { // not
NH_TAQ_ACTION_DELETE, so must be
        // disreg
ard
        if (evalSeg->taqList[0].taqAction ==
NH_TAQ_ACTION_DISREGARD) {
            // same action, so see if
            string are the same
            if (strcmp(qSeg-
>taqList[0].segString,
                evalSeg->taqList[0].segString))
                    applyFactor =
disTAQFactor; // they were different, so apply dis
factor
                }
                else // not the same
                    action, so do the absent dis
                    applyFactor =
absDisTAQFactor; // since dis takes precedence of del
                }
                else { // query had 1 TAQ, but eval had
none
                    if (qSeg->taqList[0].taqAction ==
NH_TAQ_ACTION_DELETE)
                        applyFactor = absDelTAQFactor;
                    else
                        applyFactor = absDisTAQFactor;
                }
                break;
            }
        }
    }
    else {
// one (or both) of the segments has more than 1 TAQ
value
        // First see if either has no TAQ segments. In this
case,
        // we can apply the absent factor and skip the ugly
processing
        // below
        if (numQTAQs == 0) {
            // assume the abs del factor, but look for a DIS in
the
            // eval. If we find one, set the applyFactor to

```

```

the abs dis      //      since that should take precedence
                  applyFactor = absDelTAQFactor;
                  for (int evalIndex = 0; evalIndex < numEvalTAQs;
evalIndex++)    {
                  if (evalSeg->taqList[evalIndex].taqAction ==
NH_TAQ_ACTION_DISREGARD) {
                      applyFactor = absDisTAQFactor;
                      break;
                  }
                }
            }
        else if (numEvalTAQs == 0) {
            //      assume the abs del factor, but look for a DIS in
the
            //      query.  If we find one, set the applyFactor to
the abs dis
            //      since that should take precedence
            applyFactor = absDelTAQFactor;
            for (int qIndex = 0; qIndex < numQTAQs;
qIndex++)    {
                if (qSeg->taqList[qIndex].taqAction ==
NH_TAQ_ACTION_DISREGARD) {
                    applyFactor = absDisTAQFactor;
                    break;
                }
            }
        }
        else {
            //      one segment has 2 or more TAQs, and the other
has one or more

            bool satisfiedDis = true;    //      we assume we have
satisfied the

                //      dis processing until we find
                //      a dis value, since if neither
                //      seg has a dis value, we do not

                //      apply the dis value
            bool satisfiedDel = true;    //      we assume we have
satisfied the

                //      del processing until we find
                //      a del value, since if neither
                //      seg has a del value, we do not

                //      apply the del value
            bool satisfiedAbs = true;    //      we assume we have
satisfied the

                //      abs processing.
            bool foundMatchingDis = false;
            bool foundMatchingDel = false;

            int i, j;

```



```

// go through the query segment, looking for dis
segments
    for (i = 0; i < numQTAQs; i++) {
        if (qSeg->taqList[i].taqAction ==
NH_TAQ_ACTION_DISREGARD) {
            // since we found a dis, we must find a
dis in the eval seg.
                satisfiedDis = false;
                satisfiedAbs = false;
                // look for disregards in the eval seg.
                for (j = 0; j < numEvalTAQs; j++){
                    if (evalSeg->taqList[j].taqAction ==
NH_TAQ_ACTION_DISREGARD) {
                        // found a dis, so we are
not dealing with an absent
                        // situation - see if the
segs are the same.
                            satisfiedAbs = true;
                            if (!strcmp(qSeg-
>taqList[i].segString,
                                evalSeg->taqList[j].segString)) {
                                    foundMatchingDis = true;
                                    satisfiedDis = true;
                                    break;
                                }
                            }
                        // if we get here, and the abs has not
been satisfied, we
                        // apply the abs factor, since we did
not find any dis in the
                        // eval, but did find one in the query.
                        if (satisfiedAbs == false) {
                            applyFactor = absDisTAQFactor;
                            // mark the DIS as satisfied so
that we do not
                            // re-assign the factor below
when seeing if DEL was satisfied.
                                satisfiedDis = true;
                                break;
                            }
                        else {
                            // check to see if we satisfied
the dis. If we did, we can
                                // go check out the delete stuff.
                                if (satisfiedDis == true)
                                    break;
                                }
                        }
                    }
                // end for query TAQ

// once here, we made it to the end of the query
TAQs while looking
// for disregards. This means either:
// - we found no disregards in the query
- so go on
// and see if there are any

```

```

disregards in the Eval
// - we found disregards in Q, but none
in Eval - we
// apply the absent factor, and
we're done
// - we found dis in Q, but no matching
ones in Eval - we
// apply the disregard factor,
and we're done
// - we found a matching dis in Q and
Eval - so do deletes.
// we can skip the check for
disregards in Eval, since
// we already know there is a
match.

// make sure we should continue
if (satisfiedAbs && satisfiedDis) {
//
if (foundMatchingDis == false) {
// We are in this section if the Q had
no Dis Values.
// see if there are dis values in Eval.
for (j = 0; j < numEvalTAQs; j++) {
if (evalSeg->taqList[j].taqAction ==
NH_TAQ_ACTION_DISREGARD) {
applyFactor = absDisTAQFactor;
satisfiedAbs = false;
break;
}
}
}
// see if we should still continue after
checking for reverse absent
if (satisfiedAbs) {
// when here, we got passed checking
for the dis, so we need to check for
// deletes.

// go through the query segment,
looking for del segments
for (i = 0; i < numQTAQs; i++) {
if (qSeg->taqList[i].taqAction ==
NH_TAQ_ACTION_DELETE) {
// since we found a del, we
must find a del in the eval seg.
satisfiedDel = false;
satisfiedAbs = false;
// look for deletes in the
eval seg.
for (j = 0; j < numEvalTAQs;
j++) {
if (evalSeg-
>taqList[j].taqAction == NH_TAQ_ACTION_DELETE) {
// found a del,
so we are not dealing with an absent
// situation -
see if the segs are the same.
satisfiedAbs =
true;

```

```

>taqList[i].segString,                                if (!strcmp(qSeg-
                                                         foundMatchin
                                                         satisfiedDel
                                                         break;
                                                         }
                                                         }
                                                         }
//      if we get here, and the
//      apply the abs factor,
//      eval, but did find one
if (satisfiedAbs ==
false) {
    applyFactor =
    //      mark the DEL as
    //      re-assign the
    satisfiedDel =
    break;
} else {
    //      check to see if we
    if (satisfiedDel ==
        break;
    }
} //      end for query TAQ

//      make sure we should continue
if (satisfiedAbs && satisfiedDel) {
    if (foundMatchingDel ==
false) {
        //      We are in this section
        //      see if there are del
        for (j = 0; j < numEvalTAQs;
            if (evalSeg-
                applyFactor =
                satisfiedAbs =
                break;
            }
        }
    }

```

```

    }
}

// decide the factor based on the condition that
was not satisfied // except for abs, in which case we already set the
applyFactor // above
if (satisfiedDel == false)
    applyFactor = delTAQFactor;
else if (satisfiedDis == false)
    applyFactor = disTAQFactor;
}

// apply the factor we decided on
*diScore *= applyFactor;
}

```

```

// DigraphBitmapArray.hpp : header file
//
// Class that holds the bit patterns for each possible
// digraph (AA - ZZ). We also need to account for spaces.
//
// Each bit pattern turns on just 1 bit. We basically turn
// on one bit, and shift it through the value until it reaches
// the other end, at which time we start back at the beginning
// again.
//
// Any other character are treated as spaces in our scheme,
// so we do not need to worry about them.
//
// The class supports either a 32 bit value, or a 64 bit value.
////////////////////////////////////
////

#ifndef DIGRAPHBITMAPARRAY_HPP
#define DIGRAPHBITMAPARRAY_HPP

// How many indexes do we need in our two dimensional array?
// 27 (26 letters plus a space)
#define BITMAP_ARRAY_INDEX_SIZE 27

typedef struct {
    unsigned int hiBytes;
    unsigned int lowBytes;
} bit_64_t;

class NHDigraphBitmapArray
{
// Construction
public:
    NHDigraphBitmapArray(); // standard constructor

    ~NHDigraphBitmapArray();

    unsigned int get32BitKeyForToken(char *token);

    void get64BitKeyForToken(char *token,
bit_64_t *key);

    unsigned char getNumBitsForByte(unsigned char byteVal) {return
bitTable[byteVal];}

// Implementation
protected:

    void buildBitTable();

    // the array that holds the bit map patterns for each possible
    // digraph. Each item in the array is an integer that has
    // one of its 32 bits turned on.
    unsigned
int bitMapArray32[BITMAP_ARRAY_INDEX_SIZE][BITMAP_ARRAY_INDEX_SI
ZE];

    // the array that holds the bit map patterns for each possible

```

```

    //    digraph. Each item in the array is an integer that has
    //    one of its 64 bits turned on.
    bit_64_t      bitMapArray64[BITMAP_ARRAY_INDEX_SIZE]
E][BITMAP_ARRAY_INDEX_SIZE];

    unsigned char      bitTable[256];

};

#endif

```

```

// NHDigraphBitmapArray.cpp : implementation file
//
//          3/20/98      EFB          Changed names to NH from SN

#include "NHDigraphBitmapArray.hpp"

#include <stdio.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

typedef unsigned char byte;

////////////////////////////////////
/////
// Constructor.
// Fills in the values in both of the bitMapArrays (32 bit and
// 64 bits).
NHDigraphBitmapArray::NHDigraphBitmapArray()
{
    unsigned int    bitmapValue32 = 1;
    unsigned int    bitmapValue64High = 0;
    unsigned int    bitmapValue64Low = 1;

    for (int i = 0; i < BITMAP_ARRAY_INDEX_SIZE; i++) {
        for (int j = 0; j < BITMAP_ARRAY_INDEX_SIZE; j++) {

            // assign the 32 bit value
            bitMapArray32[i][j] = bitmapValue32;

            // assign the 64 bit value
            bitMapArray64[i][j].hiBytes = bitmapValue64High;
            bitMapArray64[i][j].lowBytes = bitmapValue64Low;

            // now shift the values
            bitmapValue32 <=< 1;
            if (bitmapValue32 == 0)
                bitmapValue32 = 1;

            if (bitmapValue64Low == 0) {
                bitmapValue64High <=< 1;
                if (bitmapValue64High == 0) {
                    bitmapValue64Low = 1;
                }
            }
            else {
                bitmapValue64Low <=< 1;
                if (bitmapValue64Low == 0) {
                    bitmapValue64High = 1;
                }
            }
        }
    }

    buildBitTable();
}

```

```
NHDigraphBitmapArray::~NHDigraphBitmapArray()
{
}
```

```
void NHDigraphBitmapArray::get64BitKeyForToken(char *token, bit_64_t
*key)
```

```
{
    char *ch1;
    char *ch2;
    int index1;
    int index2;
    char spacedToken[200];

    // zero out the key we are going to return
    key->hiBytes = 0;
    key->lowBytes = 0;

    sprintf(spacedToken, " %s ", token);

    ch1 = spacedToken;
    if (*ch1 != '\0') {
        ch2 = ch1 + 1;
        while (*ch2 != '\0') {
            if (*ch1 == ' ')
                index1 = 26;
            else
                index1 = *ch1 - 'A';
            if (*ch2 == ' ')
                index2 = 26;
            else
                index2 = *ch2 - 'A';
            if ((index1 >= 0) && (index1 <
BITMAP_ARRAY_INDEX_SIZE)
                && (index2 >= 0) && (index2 <
BITMAP_ARRAY_INDEX_SIZE)) {
                key->hiBytes |=
bitMapArray64[index1][index2].hiBytes;
                key->lowBytes |=
bitMapArray64[index1][index2].lowBytes;
            }
            ch1 = ch2;
            ch2++;
        }
    }
}
```

```
unsigned int NHDigraphBitmapArray::get32BitKeyForToken(char *token)
```

```
{
    unsigned int retVal = 0;
    char *ch1;
    char *ch2;
    int index1;
    int index2;
    char spacedToken[200];
```



```

sprintf(spacedToken, " %s ", token);

ch1 = spacedToken;
if (*ch1 != '\0') {
    ch2 = ch1 + 1;
    while (*ch2 != '\0') {
        if (*ch1 == ' ')
            index1 = 26;
        else
            index1 = *ch1 - 'A';
        if (*ch2 == ' ')
            index2 = 26;
        else
            index2 = *ch2 - 'A';
        if ((index1 >= 0) && (index1 <
BITMAP_ARRAY_INDEX_SIZE)
            && (index2 >= 0) && (index2 <
BITMAP_ARRAY_INDEX_SIZE))
            retVal |= bitMapArray32[index1][index2];
        ch1 = ch2;
        ch2++;
    }
}

return retVal;
}

// build a table that says how many bits a byte value
// has turned off.
void NHDigraphBitmapArray::buildBitTable()
{
    byte tempByte;
    int i, j;
    byte bitsTurnedOff;

    for (i = 0; i < 256; i++) {
        tempByte = i;
        bitsTurnedOff = 0;
        for (j = 0; j < 8; j++) {
            if (tempByte & 1) // use this
                // if ((tempByte & 1) == 0) // use
                bitsTurnedOff++;
            tempByte >>= 1;
        }
        bitTable[i] = bitsTurnedOff;
    }
}

```

```

// File: NHCompParms.cpp
//
// Description:
//
//      Implementation to the NHCompParms class.
//
//
// History:
//
//      5/8/97      EFB      Created
//      3/3/98      EFB      Changed name of class, and move PP
parms to
//
//                                the new
NHNameParms class.
//      3/20/98     EFB      Changed names to NH from SN
//

```

```

#include <string.h>
#include <stdio.h>
#include <stdlib.h>

```

```

#include "NHCompParms.hpp"
#include "NHVariantTable.hpp"
#include "NHQAQTable.hpp"
#include "NH_variant_taq_globals.h"

```

```

NHCompParms::NHCompParms(NHParmsType parmsType)
{
    status = NH_SUCCESS;

    switch (parmsType)
    {
        case NH_PARMS_GENERIC: // default
            scoreThresh = 0.6;
            useGnLeftBias = false;
            useSnLeftBias = false;
            matchGnIntial = true;
            matchSnIntial = false;
            gnInitialScore = 0.85;
            snInitialScore = 0.0;
            gnInitialOnInitialMatchScore = 1.0;
            snInitialOnInitialMatchScore = 0.0;
            useGnVariants = true;
            useSnVariants = true;
            fnuScore = 0.60;
            nfnScore = 0.65;
            lnuScore = 0.6;
            nlnScore = 0.65;
            gnAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
            snAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
            gnAnchorFactor = 0.0;
            snAnchorFactor = 0.0;
            gnOOPSFactor = 0.6;
            snOOPSFactor = 0.6;
            disGnTAQFactor = 0.7;
            absDelGnTAQFactor = 0.9;

```

```

absDisGnTAQFactor = 0.8;
delGnTAQFactor = 0.85;
disSnTAQFactor = 0.7;
absDelSnTAQFactor = 0.9;
absDisSnTAQFactor = 0.8;
delSnTAQFactor = 0.85;
checkGnCompressedName = false;
checkSnCompressedName = false;
gnCompressedNameScore = 0.0;
snCompressedNameScore = 0.0;
scoreGnTaq = true;
scoreSnTaq = true;
gnSegmentScoreMode = NH_SEGMODE_AVG;
snSegmentScoreMode = NH_SEGMODE_AVG;
gnScoreThresh = 0.5;
snScoreThresh = 0.5;
gnWeight = 0.8;
snWeight = 1.0;
break;

case NH_PARMES_ANGLO:
scoreThresh = 0.6;
useGnLeftBias = false;
useSnLeftBias = false;
matchGnInitial = true;
matchSnInitial = false;
gnInitialScore = 0.85;
snInitialScore = 0.0;
gnInitialOnInitialMatchScore = 1.0;
snInitialOnInitialMatchScore = 0.0;
useGnVariants = true;
useSnVariants = true;
fnuScore = 0.60;
nfnScore = 0.65;
lnuScore = 0.6;
nlfnScore = 0.65;
gnAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
snAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
gnAnchorFactor = 0.0;
snAnchorFactor = 0.0;
gnOOPSFactor = 0.6;
snOOPSFactor = 0.6;
disGnTAQFactor = 0.7;
absDelGnTAQFactor = 0.9;
absDisGnTAQFactor = 0.8;
delGnTAQFactor = 0.85;
disSnTAQFactor = 0.7;
absDelSnTAQFactor = 0.9;
absDisSnTAQFactor = 0.8;
delSnTAQFactor = 0.85;
checkGnCompressedName = false;
checkSnCompressedName = false;
gnCompressedNameScore = 0.0;
snCompressedNameScore = 0.0;
scoreGnTaq = true;
scoreSnTaq = true;
gnSegmentScoreMode = NH_SEGMODE_AVG;
snSegmentScoreMode = NH_SEGMODE_AVG;
gnScoreThresh = 0.5;
snScoreThresh = 0.5;
gnWeight = 0.8;

```

```

        snWeight = 1.0;
        break;

case NH_PARMs_ARABIC:
    scoreThresh = 0.63;
    useGnLeftBias = false;
    useSnLeftBias = false;
    matchGnIntial = true;
    matchSnIntial = true;
    gnInitialScore = 0.85;
    snInitialScore = 0.85;
    gnInitialOnInitialMatchScore = 1.0;
    snInitialOnInitialMatchScore = 1.0;
    useGnVariants = false;
    useSnVariants = false;
    fnuScore = 0.60;
    nfnScore = 0.65;
    lnuScore = 0.6;
    nlnScore = 0.65;
    gnAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
    snAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
    gnAnchorFactor = 0.0;
    snAnchorFactor = 0.0;
    gnOOPSFactor = 0.7;
    snOOPSFactor = 0.9;
    disGnTAQFactor = 0.7;
    absDelGnTAQFactor = 0.9;
    absDisGnTAQFactor = 0.8;
    delGnTAQFactor = 0.85;
    disSnTAQFactor = 0.7;
    absDelSnTAQFactor = 0.9;
    absDisSnTAQFactor = 0.8;
    delSnTAQFactor = 0.85;
    checkGnCompressedName = true;
    checkSnCompressedName = true;
    gnCompressedNameScore = 0.9;
    snCompressedNameScore = 0.9;
    scoreGnTaq = true;
    scoreSnTaq = true;
    gnSegmentScoreMode = NH_SEGMODE_AVG;
    snSegmentScoreMode = NH_SEGMODE_AVG;
    gnScoreThresh = 0.63;
    snScoreThresh = 0.63;
    gnWeight = 1.0;
    snWeight = 0.8;
    break;

case NH_PARMs_CHINESE:
    scoreThresh = 0.70;
    useGnLeftBias = false;
    useSnLeftBias = false;
    matchGnIntial = false;
    matchSnIntial = false;
    gnInitialScore = 0.0;
    snInitialScore = 0.0;
    gnInitialOnInitialMatchScore = 0.0;
    snInitialOnInitialMatchScore = 0.0;
    useGnVariants = true;
    useSnVariants = true;
    fnuScore = 0.60;
    nfnScore = 0.65;

```

```

    lnuScore = 0.6;
    nlnScore = 0.65;
    gnAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
    snAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
    gnAnchorFactor = 0.0;
    snAnchorFactor = 0.0;
    gnOOPSFactor = 0.0;
    snOOPSFactor = 1.0;
    disGnTAQFactor = 0.7;
    absDelGnTAQFactor = 0.9;
    absDisGnTAQFactor = 0.8;
    delGnTAQFactor = 0.85;
    disSnTAQFactor = 0.7;
    absDelSnTAQFactor = 0.9;
    absDisSnTAQFactor = 0.8;
    delSnTAQFactor = 0.85;
    checkGnCompressedName = false;
    checkSnCompressedName = false;
    gnCompressedNameScore = 0.0;
    snCompressedNameScore = 0.0;
    scoreGnTaq = true;
    scoreSnTaq = true;
    gnSegmentScoreMode = NH_SEGMODE_LOWEST;
    snSegmentScoreMode = NH_SEGMODE_AVG;
    gnScoreThresh = 0.7;
    snScoreThresh = 0.7;
    gnWeight = 0.8;
    snWeight = 1.0;
    break;

case NH_PARMES_HISPANIC:
    scoreThresh = 0.60;
    useGnLeftBias = false;
    useSnLeftBias = false;
    matchGnInitial = true;
    matchSnInitial = true;
    gnInitialScore = 0.85;
    snInitialScore = 0.85;
    gnInitialOnInitialMatchScore = 1.0;
    snInitialOnInitialMatchScore = 1.0;
    useGnVariants = true;
    useSnVariants = true;
    fnuScore = 0.60;
    nfnScore = 0.65;
    lnuScore = 0.6;
    nlnScore = 0.65;
    gnAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
    snAnchorSegmentMode = NH_ANCHOR_SEG_FIRST;
    gnAnchorFactor = 0.0;
    snAnchorFactor = 0.70;
    gnOOPSFactor = 0.6;
    snOOPSFactor = 0.6;
    disGnTAQFactor = 0.7;
    absDelGnTAQFactor = 0.9;
    absDisGnTAQFactor = 0.8;
    delGnTAQFactor = 0.85;
    disSnTAQFactor = 0.7;
    absDelSnTAQFactor = 0.9;
    absDisSnTAQFactor = 0.8;
    delSnTAQFactor = 0.85;
    checkGnCompressedName = true;

```

```

        checkSnCompressedName = true;
        gnCompressedNameScore = 0.9;
        snCompressedNameScore = 0.9;
        scoreGnTags = true;
        scoreSnTags = true;
        gnSegmentScoreMode = NH_SEGMODE_AVG;
        snSegmentScoreMode = NH_SEGMODE_AVG;
        gnScoreThresh = 0.6;
        snScoreThresh = 0.6;
        gnWeight = 0.8;
        snWeight = 1.0;
        break;

    case NH_PARMS_KOREAN: // Parameters
        tuned for Korean names.
        scoreThresh = 0.66;
        useGnLeftBias = false;
        useSnLeftBias = false;
        matchGnInitial = false;
        matchSnInitial = false;
        gnInitialScore = 0.0;
        snInitialScore = 0.0;
        gnInitialOnInitialMatchScore = 0.0;
        snInitialOnInitialMatchScore = 0.0;
        useGnVariants = true;
        useSnVariants = true;
        fnuScore = 0.60;
        nfnScore = 0.65;
        lnuScore = 0.6;
        nlnScore = 0.65;
        gnAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
        snAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
        gnAnchorFactor = 0.0;
        snAnchorFactor = 0.0;
        gnOOPSFactor = 0.69;
        snOOPSFactor = 0.63;
        disGnTAQFactor = 0.7;
        absDelGnTAQFactor = 0.9;
        absDisGnTAQFactor = 0.8;
        delGnTAQFactor = 0.85;
        disSnTAQFactor = 0.7;
        absDelSnTAQFactor = 0.9;
        absDisSnTAQFactor = 0.8;
        delSnTAQFactor = 0.85;
        checkGnCompressedName = false;
        checkSnCompressedName = false;
        gnCompressedNameScore = 0.0;
        snCompressedNameScore = 0.0;
        scoreGnTags = true;
        scoreSnTags = true;
        gnSegmentScoreMode = NH_SEGMODE_AVG;
        snSegmentScoreMode = NH_SEGMODE_AVG;
        gnScoreThresh = 0.69;
        snScoreThresh = 0.63;
        gnWeight = 0.8;
        snWeight = 1.0;
        break;

    case NH_PARMS_RUSSIAN: // Parameters
        tuned for Russian names.
        scoreThresh = 0.61;

```

```

        useGnLeftBias = false;
        useSnLeftBias = true;
        matchGnIntial = true;
        matchSnIntial = true;
        gnInitialScore = 0.85;
        snInitialScore = 0.85;
        gnInitialOnInitialMatchScore = 1.0;
        snInitialOnInitialMatchScore = 1.0;
        useGnVariants = false;
        useSnVariants = false;
        fnuScore = 0.60;
        nfnScore = 0.65;
        lnuScore = 0.6;
        nlnScore = 0.65;
        gnAnchorSegmentMode = NH_ANCHOR_SEG_FIRST;
        snAnchorSegmentMode = NH_ANCHOR_SEG_NONE;
        gnAnchorFactor = 0.60;
        snAnchorFactor = 0.00;
        gnOOPSFactor = 0.65;
        snOOPSFactor = 0.8;
        disGnTAQFactor = 0.7;
        absDelGnTAQFactor = 0.9;
        absDisGnTAQFactor = 0.8;
        delGnTAQFactor = 0.85;
        disSnTAQFactor = 0.7;
        absDelSnTAQFactor = 0.9;
        absDisSnTAQFactor = 0.8;
        delSnTAQFactor = 0.85;
        checkGnCompressedName = false;
        checkSnCompressedName = false;
        gnCompressedNameScore = 0.0;
        snCompressedNameScore = 0.0;
        gnSegmentScoreMode = NH_SEGMODE_HIGHEST;
        snSegmentScoreMode = NH_SEGMODE_AVG;
        gnScoreThresh = 0.6;
        snScoreThresh = 0.62;
        gnWeight = 0.8;
        snWeight = 1.0;
        break;
    }
    // end of switch
}

```

```

NHCompParms::NHCompParms(istream &inStream)

```

```

{
    int    compParmsVersion;

    if (inStream.good()) {
        inStream.read((char *)&compParmsVersion, sizeof(int));

        inStream.read((char *)&scoreThresh, sizeof(double));
        inStream.read((char *)&useGnLeftBias, sizeof(bool));
        inStream.read((char *)&useSnLeftBias, sizeof(bool));
        inStream.read((char *)&matchGnIntial, sizeof(bool));
        inStream.read((char *)&matchSnIntial, sizeof(bool));
        inStream.read((char *)&gnInitialScore, sizeof(double));
        inStream.read((char *)&snInitialScore, sizeof(double));
        inStream.read((char *)&useGnVariants, sizeof(bool));
        inStream.read((char *)&useSnVariants, sizeof(bool));
        inStream.read((char *)&fnuScore, sizeof(double));
        inStream.read((char *)&nfnScore, sizeof(double));
    }
}

```

```

        inStream.read((char *)&lnuScore, sizeof(double));
        inStream.read((char *)&lnlnScore, sizeof(double));

        inStream.read((char *)&gnSegmentScoreMode,
sizeof(NHSegScoreMode));
        inStream.read((char *)&snSegmentScoreMode,
sizeof(NHSegScoreMode));
        inStream.read((char *)&gnAnchorSegmentMode,
sizeof(NHAnchorSegMode));
        inStream.read((char *)&snAnchorSegmentMode,
sizeof(NHAnchorSegMode));

        inStream.read((char *)&gnAnchorFactor, sizeof(double));
        inStream.read((char *)&snAnchorFactor, sizeof(double));
        inStream.read((char *)&gnOOPSFactor, sizeof(double));
        inStream.read((char *)&snOOPSFactor, sizeof(double));

        inStream.read((char *)&scoreGnTaq, sizeof(bool));
        inStream.read((char *)&scoreSnTaq, sizeof(bool));

        inStream.read((char *)&absDelGnTAQFactor, sizeof(double));
        inStream.read((char *)&absDisGnTAQFactor, sizeof(double));
        inStream.read((char *)&absDelSnTAQFactor, sizeof(double));
        inStream.read((char *)&absDisSnTAQFactor, sizeof(double));
        inStream.read((char *)&delGnTAQFactor, sizeof(double));
        inStream.read((char *)&delSnTAQFactor, sizeof(double));
        inStream.read((char *)&disGnTAQFactor, sizeof(double));
        inStream.read((char *)&disSnTAQFactor, sizeof(double));

        inStream.read((char *)&checkGnCompressedName, sizeof(bool));
        inStream.read((char *)&checkSnCompressedName, sizeof(bool));

        inStream.read((char *)&gnCompressedNameScore,
sizeof(double));
        inStream.read((char *)&snCompressedNameScore,
sizeof(double));

        inStream.read((char *)&gnScoreThresh, sizeof(double));
        inStream.read((char *)&snScoreThresh, sizeof(double));

        inStream.read((char *)&gnWeight, sizeof(double));
        inStream.read((char *)&snWeight, sizeof(double));

        inStream.read((char *)&gnInitialOnInitialMatchScore,
sizeof(double));
        inStream.read((char *)&snInitialOnInitialMatchScore,
sizeof(double));

        status = NH_SUCCESS;
    }
    else
        status = NH_COMP_PARMS_BAD_STREAM_ON_CONSTRUCT;
}

NHCompParms::~NHCompParms()
{
}

NHReturnCode NHCompParms::archiveData(ostream &ostream)

```



```

{
// comp parms file version history
// 1.0 - first version
int compParmsVersion = 1;
NHReturnCode rc = NH_SUCCESS;

if (outStream.good()) {
    outStream.write((char *)&compParmsVersion, sizeof(int));

    outStream.write((char *)&scoreThresh, sizeof(double));
    outStream.write((char *)&useGnLeftBias, sizeof(bool));
    outStream.write((char *)&useSnLeftBias, sizeof(bool));
    outStream.write((char *)&matchGnIntial, sizeof(bool));
    outStream.write((char *)&matchSnIntial, sizeof(bool));
    outStream.write((char *)&gnInitialScore, sizeof(double));
    outStream.write((char *)&snInitialScore, sizeof(double));
    outStream.write((char *)&useGnVariants, sizeof(bool));
    outStream.write((char *)&useSnVariants, sizeof(bool));
    outStream.write((char *)&fnuScore, sizeof(double));
    outStream.write((char *)&nfnScore, sizeof(double));
    outStream.write((char *)&lnuScore, sizeof(double));
    outStream.write((char *)&nlnScore, sizeof(double));

    outStream.write((char *)&gnSegmentScoreMode,
sizeof(NHSegScoreMode));
    outStream.write((char *)&snSegmentScoreMode,
sizeof(NHSegScoreMode));
    outStream.write((char *)&gnAnchorSegmentMode,
sizeof(NHAnchorSegMode));
    outStream.write((char *)&snAnchorSegmentMode,
sizeof(NHAnchorSegMode));

    outStream.write((char *)&gnAnchorFactor, sizeof(double));
    outStream.write((char *)&snAnchorFactor, sizeof(double));
    outStream.write((char *)&gnOOPSFactor, sizeof(double));
    outStream.write((char *)&snOOPSFactor, sizeof(double));

    outStream.write((char *)&scoreGnTaq, sizeof(bool));
    outStream.write((char *)&scoreSnTaq, sizeof(bool));

    outStream.write((char *)&absDelGnTAQFactor, sizeof(double));
    outStream.write((char *)&absDisGnTAQFactor, sizeof(double));
    outStream.write((char *)&absDelSnTAQFactor, sizeof(double));
    outStream.write((char *)&absDisSnTAQFactor, sizeof(double));
    outStream.write((char *)&delGnTAQFactor, sizeof(double));
    outStream.write((char *)&delSnTAQFactor, sizeof(double));
    outStream.write((char *)&disGnTAQFactor, sizeof(double));
    outStream.write((char *)&disSnTAQFactor, sizeof(double));

    outStream.write((char *)&checkGnCompressedName,
sizeof(bool));
    outStream.write((char *)&checkSnCompressedName,
sizeof(bool));

    outStream.write((char *)&gnCompressedNameScore,
sizeof(double));
    outStream.write((char *)&snCompressedNameScore,
sizeof(double));

    outStream.write((char *)&gnScoreThresh, sizeof(double));
    outStream.write((char *)&snScoreThresh, sizeof(double));

```

```

        outputStream.write((char *)&gnWeight, sizeof(double));
        outputStream.write((char *)&snWeight, sizeof(double));

        outputStream.write((char *)&gnInitialOnInitialMatchScore,
sizeof(double));
        outputStream.write((char *)&snInitialOnInitialMatchScore,
sizeof(double));
    }
    else
        rc = NH_COMP_PARMS_BAD_STREAM_ON_ARCHIVE;

    return rc;
}

```

```

NHReturnCode      NHCompParms::setScoreThresh(double aThresh)
{
    NHReturnCode      errorCode;

    if ((aThresh < 0.0) || (aThresh > 1.0))
        errorCode = NH_INVALID_SCORE_THRESH;
    else {
        scoreThresh = aThresh;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

```

```

void NHCompParms::setUseGnLeftBias(bool aBool)
{
    useGnLeftBias = aBool;
}

```

```

void NHCompParms::setUseSnLeftBias(bool aBool)
{
    useSnLeftBias = aBool;
}

```

```

void NHCompParms::setMatchGnIntial(bool aBool)
{
    matchGnIntial = aBool;
}

```

```

void NHCompParms::setMatchSnIntial(bool aBool)
{
    matchSnIntial = aBool;
}

```

```

NHReturnCode      NHCompParms::setGnInitialScore(double aScore)
{
    NHReturnCode      errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_GN_INIT_SCORE;
    else {

```

```

        gnInitialScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParms::setSnInitialScore(double aScore)
{
    NHReturnCode      errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_NH_INIT_SCORE;
    else {
        snInitialScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParms::setGnInitialOnInitialMatchScore(double
aScore)
{
    NHReturnCode      errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_GN_INIT_ON_INIT_MATCH_SCORE;
    else {
        gnInitialOnInitialMatchScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParms::setSnInitialOnInitialMatchScore(double
aScore)
{
    NHReturnCode      errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_NH_INIT_ON_INIT_MATCH_SCORE;
    else {
        snInitialOnInitialMatchScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

void NHCompParms::setUseGnVariants(bool aBool)
{
    useGnVariants = aBool;
}

```

```

void NHCompParms::setUseSnVariants(bool aBool)
{
    useSnVariants = aBool;
}

```

```

NHReturnCode NHCompParms::setNFNScore(double aScore)
{
    NHReturnCode    errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_NFN_SCORE;
    else {
        nfnScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

```

```

NHReturnCode NHCompParms::setFNUScore(double aScore)
{
    NHReturnCode    errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_FNU_SCORE;
    else {
        fnuScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

```

```

NHReturnCode NHCompParms::setNLNScore(double aScore)
{
    NHReturnCode    errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_NLN_SCORE;
    else {
        nlNScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

```

```

NHReturnCode NHCompParms::setLNUScore(double aScore)
{
    NHReturnCode    errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_LNU_SCORE;
    else {
        lnuScore = aScore;
        errorCode = NH_SUCCESS;
    }
}

```

```

        return errorCode;
    }

NHReturnCode      NHCompParams::setGnScoreThresh(double aThresh)
{
    NHReturnCode      errorCode;

    if ((aThresh < 0.0) || (aThresh > 1.0))
        errorCode = NH_INVALID_GN_THRESH;
    else {
        gnScoreThresh = aThresh;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setSnScoreThresh(double aThresh)
{
    NHReturnCode      errorCode;

    if ((aThresh < 0.0) || (aThresh > 1.0))
        errorCode = NH_INVALID_NH_THRESH;
    else {
        snScoreThresh = aThresh;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setGnWeight(double aWeight)
{
    NHReturnCode      errorCode;

    if ((aWeight < 0.0) || (aWeight > 1.0))
        errorCode = NH_INVALID_GN_WEIGHT;
    else {
        gnWeight = aWeight;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setSnWeight(double aWeight)
{
    NHReturnCode      errorCode;

    if ((aWeight < 0.0) || (aWeight > 1.0))
        errorCode = NH_INVALID_NH_WEIGHT;
    else {
        snWeight = aWeight;
        errorCode = NH_SUCCESS;
    }
}

```

```

        return errorCode;
    }

void NHCompParms::setGnSegmentScoreMode(NHSegScoreMode aMode)
{
    gnSegmentScoreMode = aMode;
}

void NHCompParms::setSnSegmentScoreMode(NHSegScoreMode aMode)
{
    snSegmentScoreMode = aMode;
}

void NHCompParms::setGnAnchorSegmentMode(NHAnchorSegMode anAnchorMode)
{
    gnAnchorSegmentMode = anAnchorMode;
}

void NHCompParms::setSnAnchorSegmentMode(NHAnchorSegMode anAnchorMode)
{
    snAnchorSegmentMode = anAnchorMode;
}

NHReturnCode NHCompParms::setGnAnchorFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_GN_ANCHOR_FACTOR;
    else {
        gnAnchorFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode NHCompParms::setSnAnchorFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_NH_ANCHOR_FACTOR;
    else {
        snAnchorFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode NHCompParms::setGnOOPSFactor(double aFactor)
{
    NHReturnCode      errorCode;

```

```

        if ((aFactor < 0.0) || (aFactor > 1.0))
            errorCode = NH_INVALID_GN_OOPS_FACTOR;
        else {
            gnOOPSFactor = aFactor;
            errorCode = NH_SUCCESS;
        }

        return errorCode;
    }

NHReturnCode      NHCompParams::setSnOOPSFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_NH_OOPS_FACTOR;
    else {
        snOOPSFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setAbsDelGnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_ABS_DEL_GN_TAQ_FACTOR;
    else {
        absDelGnTAQFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setAbsDisGnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_ABS_DIS_GN_TAQ_FACTOR;
    else {
        absDisGnTAQFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setAbsDelSnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

```

```

        if ((aFactor < 0.0) || (aFactor > 1.0))
            errorCode = NH_INVALID_ABS_DEL_NH_TAQ_FACTOR;
        else {
            absDelSnTAQFactor = aFactor;
            errorCode = NH_SUCCESS;
        }

        return errorCode;
    }

NHReturnCode      NHCompParams::setAbsDisSnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_ABS_DIS_NH_TAQ_FACTOR;
    else {
        absDisSnTAQFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setDelGnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_DEL_GN_TAQ_FACTOR;
    else {
        delGnTAQFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setDelSnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_DEL_NH_TAQ_FACTOR;
    else {
        delSnTAQFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setDisGnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))

```



```

        errorCode = NH_INVALID_DIS_GN_TAQ_FACTOR;
    else {
        disGnTAQFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

NHReturnCode      NHCompParams::setDisSnTAQFactor(double aFactor)
{
    NHReturnCode      errorCode;

    if ((aFactor < 0.0) || (aFactor > 1.0))
        errorCode = NH_INVALID_DIS_NH_TAQ_FACTOR;
    else {
        disSnTAQFactor = aFactor;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

void NHCompParams::setScoreGnTAQs(bool aBool)
{
    scoreGnTaqS = aBool;
}

void NHCompParams::setScoreSnTAQs(bool aBool)
{
    scoreSnTaqS = aBool;
}

void      NHCompParams::setCheckGnCompressedName(bool aBool)
{
    checkGnCompressedName = aBool;
}

void      NHCompParams::setCheckSnCompressedName(bool aBool)
{
    checkSnCompressedName = aBool;
}

NHReturnCode      NHCompParams::setGnCompressedNameScore(double
aScore)
{
    NHReturnCode      errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_GN_COMPRESSED_NAME_SCORE;
    else {
        gnCompressedNameScore = aScore;
        errorCode = NH_SUCCESS;
    }
}

```

```

        return errorCode;
    }

NHReturnCode      NHCompParams::setSnCompressedNameScore(double
aScore)
{
    NHReturnCode      errorCode;

    if ((aScore < 0.0) || (aScore > 1.0))
        errorCode = NH_INVALID_NH_COMPRESSED_NAME_SCORE;
    else {
        snCompressedNameScore = aScore;
        errorCode = NH_SUCCESS;
    }

    return errorCode;
}

bool      NHCompParams::operator==(NHCompParams &other)
{
    bool rc;

    rc = ((scoreThresh == other.scoreThresh) &&
        (useGnLeftBias == other.useGnLeftBias) &&
        (useSnLeftBias == other.useSnLeftBias) &&
        (matchGnInitial == other.matchGnInitial) &&
        (matchSnInitial == other.matchSnInitial) &&
        (gnInitialScore == other.gnInitialScore) &&
        (snInitialScore == other.snInitialScore) &&
        (useGnVariants == other.useGnVariants) &&
        (useSnVariants == other.useSnVariants) &&
        (fnuScore == other.fnuScore) &&
        (nfnScore == other.nfnScore) &&
        (lnuScore == other.lnuScore) &&
        (nlfnScore == other.nlfnScore) &&
        (gnSegmentScoreMode == other.gnSegmentScoreMode)
    &&
        (snSegmentScoreMode == other.snSegmentScoreMode)
    &&
        (gnAnchorSegmentMode ==
other.gnAnchorSegmentMode) &&
        (snAnchorSegmentMode ==
other.snAnchorSegmentMode) &&
        (gnAnchorFactor == other.gnAnchorFactor) &&
        (snAnchorFactor == other.snAnchorFactor) &&
        (gnOOPSFactor == other.gnOOPSFactor) &&
        (snOOPSFactor == other.snOOPSFactor) &&
        (gnWeight == other.gnWeight) &&
        (snWeight == other.snWeight) &&
        (gnScoreThresh == other.gnScoreThresh) &&
        (snScoreThresh == other.snScoreThresh) &&
        (scoreGnTaq == other.scoreGnTaq) &&
        (scoreSnTaq == other.scoreSnTaq) &&
        (absDelGnTAQFactor == other.absDelGnTAQFactor)
    &&
        (absDisGnTAQFactor == other.absDisGnTAQFactor)
    &&
        (absDelSnTAQFactor == other.absDelSnTAQFactor)
    &&

```

```

        (absDisSnTAQFactor == other.absDisSnTAQFactor)
    &&
        (delGnTAQFactor == other.delGnTAQFactor) &&
        (delSnTAQFactor == other.delSnTAQFactor) &&
        (disGnTAQFactor == other.disGnTAQFactor) &&
        (disSnTAQFactor == other.disSnTAQFactor) &&
        (checkGnCompressedName ==
other.checkGnCompressedName) &&
        (checkSnCompressedName ==
other.checkSnCompressedName) &&
        (gnCompressedNameScore ==
other.gnCompressedNameScore) &&
        (snCompressedNameScore ==
other.snCompressedNameScore) &&
        (gnInitialOnInitialMatchScore ==
other.gnInitialOnInitialMatchScore) &&
        (snInitialOnInitialMatchScore ==
other.snInitialOnInitialMatchScore));
    return rc;
}

```

```

NHReturnCode      NHCompParms::getStatus()
{
    return status;
}

```

```
//      File:  NH_variant_taq_globals.h
//
//      Description:
//
//          Functions to manage the global variant and TAQ resources.
//          We manage the TAQ and variant tables as global resources
//          so that each SNCompParms object does not need to create its
//          own copy of them.  We provide these global functions so that
//          we can control the variables in one location.
//
//
//      History:
//
//          9/08/97      EFB      Created
//          3/20/98      EFB      Changed names to NH from SN
//
//
#ifdef NH_VARIANT_TAQ_GLOBALS_DEFFED
#define NH_VARIANT_TAQ_GLOBALS_DEFFED

#include      "NH_culture_codes.h"

//      function to return pointers to the global SN and GN Variant Tables
NHVariantTable      *NH_getVariantTable(NH_VARIANT_TABLE_TYPES
variantTableType);

NHTAQTable      *NH_getTAQTable();

#endif
```

```

// File: NH_variant_taq_globals.cpp
//
// Description:
//
// Functions to manage the global variant and TAQ resources.
// We manage the TAQ and variant tables as global resources
// so that each NHCompParms object does not need to create its
// own copy of them. We provide these global functions so that
// we can control the variables in one location.
//
// We should provide some sort of thread protection around
these
// resources to make sure that two competing threads do not
attempt
// to grab these resources during creation time. How can we do
this
// portably?.
//
//
// History:
//
// 9/08/97 EFB Created
// 3/20/98 EFB Changed names to NH from SN
//

```

```

#include <string.h>

```

```

#include "NH_util.hpp"
#include "NHVariantTable.hpp"
#include "NHTAQTable.hpp"

#include "NH_variant_taq_globals.h"

```

```

// define SN and GN variant tables
NHVariantTable *NH_snVariantTable = NULL;
NHVariantTable *NH_gnVariantTable = NULL;

```

```

// define a single TAQ table
NHTAQTable *NH_taqTable = NULL;

```

```

// functions to create and return pointers to the tables

```

```

NHVariantTable *NH_getVariantTable(NH_VARIANT_TABLE_TYPES
variantTableType)
{
    NHVariantTable *tablePtr;
    NHVariantTable **tablePtrPtr = NULL;

    switch (variantTableType) {
        case NH_SURNAME_VARIANTS:
            tablePtr = NH_snVariantTable;
            tablePtrPtr = &NH_snVariantTable;
            break;
        case NH_GIVENNAME_VARIANTS:
            tablePtr = NH_gnVariantTable;
            tablePtrPtr = &NH_gnVariantTable;
            break;
    }
}

```

```

        default:
            tablePtr = NULL;
    }
    if (tablePtr == NULL) {
        tablePtr = new
NHVariantTable(variantTableType);           // create the table
        if (tablePtrPtr != NULL)
            *tablePtrPtr = tablePtr;         // assign the global
variable
    }
    return    tablePtr;
}

```

```

NHQAQTable *NH_getQAQTable()
{
    if (NH_taqTable == NULL) {
        NH_taqTable = new
NHQAQTable(NH_PRODUCTION_TAQ_TABLE);        // create the table
    }
    return    NH_taqTable;
}

```

```

//      File:  NH_util.cpp
//
//      Description:
//
//          Implementation of various utility functions used in the
SNAPI
//
//
//      History:
//
//          5/15/97      EFB      Created
//          3/20/98      EFB      Changed names to NH from SN
//

#include    <string.h>

#include    "NH_util.hpp"
#include    "NHCompParms.hpp"

//      function to remove leading and trailing spaces from a string
//      in place.
//      Strips the string at either end or both ends.
//      Stripchars specify the characters that should
//      be stripped.  We start by seeing if they want the
//      trailing chars stripped, which is easy.  We simply
//      work backwards from the end of the string, looking for
//      the first non-strippable character, and terminate the
//      string just past that character.  Then if they wanted
//      leading chars stripped, we work forwards to the first
//      non-strippable char, and then move that and each following
//      char to the beginning of the string.
void NH_strip(char *aString)
{
    char *end_point;
    char *ch;
    int len;

    if ((len = strlen(aString)) != 0) { // if there is a string
        // start at end
        end_point = aString + len - 1;

        // and work back till we get a non-space or get to
        // the begining of our string, chopping off what's left.
        // Also make sure we don't zoom right past the beginning of
the
        // string.
        for (; strchr(NH_DEFAULT_WHITESPACE, *end_point) != NULL &&
end_point != aString; end_point--)
            ;
        // if string was all whitespace
        if ((end_point == aString) && strchr(NH_DEFAULT_WHITESPACE,
*aString) != NULL)
            *aString = EOS; // erase it all, and we're done,
could return here
        else
            *(end_point + 1) = EOS; // just chop off excess
    }
}

```

blanks

```
    // make sure there is still a string, since it might
    // have been stripped entirely above.
    if (*aString) {
        // now find first non space. we know string has at
least one
        // nonwhite space, so we don't have to check for
NULL.
        for (ch = aString; strchr(NH_DEFAULT_WHITESPACE, *ch)
!= NULL; ch++)
            ;
        if (ch != aString) { // if there were leading spaces,
move the block back
            char *target = aString;
            while (*ch != EOS) {
                *target = *ch;
                target++;
                ch++;
            }
            // and get the null char also
            *target = *ch;
        } // end if (are there leading spaces?)
    } // end if (and text left?)
} // end (is there a string at all ?)
}
```

```
char *    NH_strrchr(char *stringStart, char *searchPos, char
searchChar)
{
    while (1) {
        if (*searchPos == searchChar)
            break;
        if (searchPos == stringStart) {
            searchPos = NULL; // string not found, so
return NULL
            break;
        }
        searchPos--;
    }
    return searchPos;
}
```



```

//
// File: NH_queens_arrays.hpp
//
// Description:
//
// Contains global definitions and declarations for the valid
// combinations of indexes for the best score calculation
//
// History:
//
// 6/4/97 EFB Created
// 3/20/98 EFB Changed names to NH from SN
//

typedef unsigned char byte;

byte twoByTwo[] = {1, 0,
                  0, 1};

byte twoByThree[] = { 1, 2,
1, 0,
2, 1,
2, 0,
0, 1,
0, 2};

byte twoByFour[] = { 1, 2,
1, 3,
1, 0,
2, 1,
2, 3,
2, 0,
3, 1,
3, 2,
3, 0,
0, 1,
0, 2,
0, 3};

byte twoByFive[] = { 1, 2,
1, 3,

```

1, 4,  
1, 0,  
2, 1,  
2, 3,  
2, 4,  
2, 0,  
3, 1,  
3, 2,  
3, 4,  
3, 0,  
4, 1,  
4, 2,  
4, 3,  
4, 0,  
0, 1,  
0, 2,  
0, 3,  
0, 4};

byte threeByThree[] = { 1, 2, 0,

1, 0, 2,

2, 1, 0,

2, 0, 1,

0, 1, 2,

0, 2, 1});

byte threeByFour[] = { 1, 2, 3,

1, 2, 0,

1, 3, 2,

1, 3, 0,

1, 0, 2,

1, 0, 3,

```
2, 1, 3,  
2, 1, 0,  
2, 3, 1,  
2, 3, 0,  
2, 0, 1,  
2, 0, 3,  
3, 1, 2,  
3, 1, 0,  
3, 2, 1,  
3, 2, 0,  
3, 0, 1,  
3, 0, 2,  
0, 1, 2,  
0, 1, 3,  
0, 2, 1,  
0, 2, 3,  
0, 3, 1,  
0, 3, 2};  
byte threeByFive[] = { 1, 2, 3,  
1, 2, 4,  
1, 2, 0,  
1, 3, 2,  
1, 3, 4,  
1, 3, 0,  
1, 4, 2,  
1, 4, 3,  
1, 4, 0,  
1, 0, 2,  
1, 0, 3,  
1, 0, 4,  
2, 1, 3,
```

2, 1, 4,  
2, 1, 0,  
2, 3, 1,  
2, 3, 4,  
2, 3, 0,  
2, 4, 1,  
2, 4, 3,  
2, 4, 0,  
2, 0, 1,  
2, 0, 3,  
2, 0, 4,  
3, 1, 2,  
3, 1, 4,  
3, 1, 0,  
3, 2, 1,  
3, 2, 4,  
3, 2, 0,  
3, 4, 1,  
3, 4, 2,  
3, 4, 0,  
3, 0, 1,  
3, 0, 2,  
3, 0, 4,  
4, 1, 2,  
4, 1, 3,  
4, 1, 0,  
4, 2, 1,  
4, 2, 3,  
4, 2, 0,  
4, 3, 1,

```
4, 3, 2,
4, 3, 0,
4, 0, 1,
4, 0, 2,
4, 0, 3,
0, 1, 2,
0, 1, 3,
0, 1, 4,
0, 2, 1,
0, 2, 3,
0, 2, 4,
0, 3, 1,
0, 3, 2,
0, 3, 4,
0, 4, 1,
0, 4, 2,
0, 4, 3};
byte fourByFour[] = { 1, 2, 3, 0,
1, 2, 0, 3,
1, 3, 0, 2,
1, 3, 2, 0,
1, 0, 2, 3,
1, 0, 3, 2,
2, 1, 3, 0,
2, 1, 0, 3,
2, 3, 1, 0,
2, 3, 0, 1,
2, 0, 1, 3,
2, 0, 3, 1,
3, 1, 2, 0,
3, 1, 0, 2,
```

3, 2, 1, 0,

3, 2, 0, 1,

3, 0, 1, 2,

3, 0, 2, 1,

0, 1, 2, 3,

0, 1, 3, 2,

0, 2, 1, 3,

0, 2, 3, 1,

0, 3, 1, 2,

0, 3, 2, 1};

byte fourByFive[] = { 1, 2, 3, 4,

1, 2, 3, 0,

1, 2, 4, 3,

1, 2, 4, 0,

1, 2, 0, 3,

1, 2, 0, 4,

1, 3, 2, 4,

1, 3, 2, 0,

1, 3, 4, 2,

1, 3, 4, 0,

1, 3, 0, 2,

1, 3, 0, 4,

1, 4, 2, 3,

1, 4, 2, 0,

1, 4, 3, 2,

1, 4, 3, 0,

1, 4, 0, 2,

1, 4, 0, 3,

1, 0, 2, 3,

1, 0, 2, 4,

1, 0, 3, 2,  
1, 0, 3, 4,  
1, 0, 4, 2,  
1, 0, 4, 3,  
2, 1, 3, 4,  
2, 1, 3, 0,  
2, 1, 4, 3,  
2, 1, 4, 0,  
2, 1, 0, 3,  
2, 1, 0, 4,  
2, 3, 1, 4,  
2, 3, 1, 0,  
2, 3, 4, 1,  
2, 3, 4, 0,  
2, 3, 0, 1,  
2, 3, 0, 4,  
2, 4, 1, 3,  
2, 4, 1, 0,  
2, 4, 3, 1,  
2, 4, 3, 0,  
2, 4, 0, 1,  
2, 4, 0, 3,  
2, 0, 1, 3,  
2, 0, 1, 4,  
2, 0, 3, 1,  
2, 0, 3, 4,  
2, 0, 4, 1,  
2, 0, 4, 3,  
3, 2, 1, 4,  
3, 2, 1, 0,  
3, 2, 4, 1,

3, 2, 4, 0,  
3, 2, 0, 1,  
3, 2, 0, 4,  
3, 1, 2, 4,  
3, 1, 2, 0,  
3, 1, 4, 2,  
3, 1, 4, 0,  
3, 1, 0, 2,  
3, 1, 0, 4,  
3, 4, 2, 1,  
3, 4, 2, 0,  
3, 4, 1, 2,  
3, 4, 1, 0,  
3, 4, 0, 2,  
3, 4, 0, 1,  
3, 0, 2, 1,  
3, 0, 2, 4,  
3, 0, 1, 2,  
3, 0, 1, 4,  
3, 0, 4, 2,  
3, 0, 4, 1,  
4, 2, 3, 1,  
4, 2, 3, 0,  
4, 2, 1, 3,  
4, 2, 1, 0,  
4, 2, 0, 3,  
4, 2, 0, 1,  
4, 3, 2, 1,  
4, 3, 2, 0,  
4, 3, 1, 2,



```
/* Generated by VariantManager */  
addVariant("ANN","ANITA",0.85,"E ");  
addVariant("ANN","ANA",0.85,"E ");  
addVariant("ANN","ANNIE",0.90,"E ");  
addVariant("ANN","ANNA",0.85,"E ");  
addVariant("ANN","ANNE",0.95,"E ");  
addVariant("ANN","ANNETTE",0.85,"E ");
```

```
/* Generated by VariantManager */  
addVariant("SON","SWUN",0.95,"C ");  
addVariant("SON","SHON",0.95,"K ");  
addVariant("SON","SOHN",0.95,"K ");
```

```
/* Generated by TAQManager */  
addTAQValue("SEÑORITA",'T','N','X','X',"G ");
```